**Authentication and Authorization in Web API**

In this article, I am going to discuss **Authentication and Authorization in Web API.**Here I will give you an overview of Authentication and Authorization in Web API and from the next article onwards, we will discuss the practical implementation of Authentication and Authorization in ASP.NET Web API with examples.

Once you create a Web API Service, then the most important thing that you need to take care of is **security** means you need to control access to your Web API Services. So let’s start the discussion with the definition of **Authentication** and **Authorization**

**Authentication**is the process of identifying the user. For example, one user let’s say James logs in with his username and password, and the server uses his username and password to authenticate James.

**Authorization**is the process of deciding whether the authenticated user is allowed to perform an action on a specific resource (Web API Resource) or not. For example, James (who is an authenticated user) has the permission to get a resource but does not have the permission to create a resource.

**Authentication in Web API**

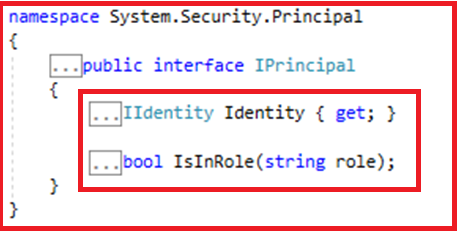
The Web API Service assumes that the authentication process should happen in the host Server and we generally host the Web API Service at IIS. The IIS Server uses the **HTTP modules** for checking the authentication of a user. You can configure your project to use any of the built-in authentication modules which are available in IIS or ASP.NET, or you can also create your own HTTP module to perform custom authentication.

When the host (IIS Server) authenticates the user, it generally creates a principal object (i.e. **IPrincipal** object) under which the code is going to run. So, once the Principal object (**IPrincipal** object) is created, then the host (i.e. IIS Server) attaches that **principal object** to the **current thread** by setting **Thread.CurrentPrincipal**.

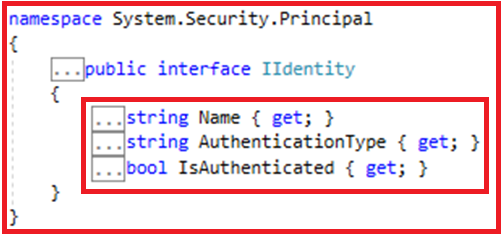
If you are confused at the moment about how the Principal object is created and how the principal object is attached to the current thread, then don’t worry we will discuss all these things in greater detail in our upcoming articles. In this article, I am just going to give you an overview of how authentication and authorization happen in Web API services.

**Understanding Principal Object**

The Principal object contains two things one is the **Identity** object which actually contains the information about the user and the other one is the **IsInRole** property which is a boolean property and this property is set to true if the user is assigned with any roles else false. The following diagram shows the IPrincipal interface definition.



Let’s have a look at the Identity interface definition which contains the user’s information.



The Identity Object which is a property of Principal Object contains three properties i.e. **Name** (string type), **AuthenticationType**(string type), and **IsAuthenticated** (boolean type). If the user is authenticated, then the **Identity.IsAuthenticated** property will return **true**else **false**. The **Name** property of the Identity object will store the name of Identity, generally, identity is nothing but the logged-in username. Similarly, the **AuthenticationType** property returns the type of authentication used to identify the user.

The Identity interface is generally implemented by the **GenericIdentity**and **WindowsIdentity**classes. We will discuss how these classes implement the **IIdentity interface** in our upcoming article.

**HTTP Message Handlers for Authentication in Web API**

Instead of using the host (i.e. IIS Server where the Web API service is hosted) for authentication, you can also write the authentication logic into a custom **HTTP Message** **Handler**. In that case, the **HTTP Message Handler** is going to check the incoming HTTP request for authenticating the user and then set the Principal Object.

**Differences HTTP Message Handler over HTTP Module:**

An **HTTP Module** sees all the incoming requests that go through the ASP.NET pipeline whereas a message handler only sees the incoming requests which are routed to the Web API Service.

It is also possible to select a specific **HTTP Message Handler** and then you can use that specific **HTTP Message Handler** for authentication for a specific route. The **HTTP Modules** are specific to IIS whereas the **HTTP Message Handlers** can be used with both **web-hosting** (within a server) and **self-hosting** (within an application).

The HTTP Modules participate in IIS logging, auditing, and so on. Generally, if you don’t want to support self-hosting, then HTTP Module is a better option but if you want to support self-hosting then HTTP Message Handler is a better option.

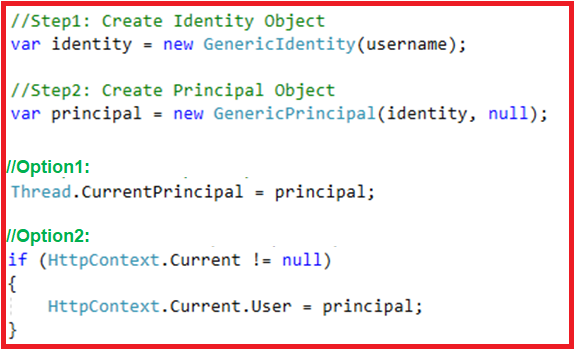
**Setting the Principal Object**

If  you are going to implement your own custom logic for authenticating the user then you can set the principal object at two places which are as follows:

**Thread.CurrentPrincipal**. This is the standard way to set the thread’s principal in .NET.

**HttpContext.Current.User**. This property is specific to ASP.NET.

The following image shows how to create and set the principal object with the current thread. Here I am showing you both the options to set the Principal object.

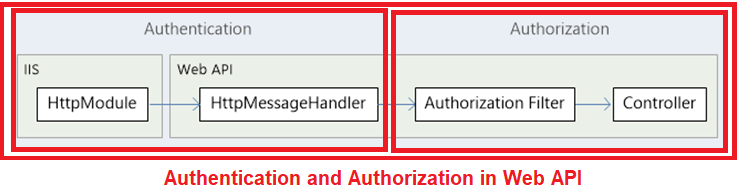


If you are going to host the Web API service in IIS, then you need to set the principal object in both places because of the security concerns i.e. security becomes inconsistent. In the case of Self-hosting the **HttpContext.Current** value is null. To ensure your code is host-agnostic (i.e. to support both web hosting and self-hosting), you need to check for null before assigning the Principal object to the **HttpContext.Current** as shown in the above image.

**Authorization in Web API**

The Authorization Process is going to happen before executing the Controller Action Method which provides you the flexibility to decide whether you want to grant access to that resource or not.

We can implement this in ASP.NET Web API by using the Authorization filters which will be executed before the controller action method executed. So, if the request is not authorized for that specific resource, then the filter returns an error response to the client without executing the controller action method. The following diagram explains the above.



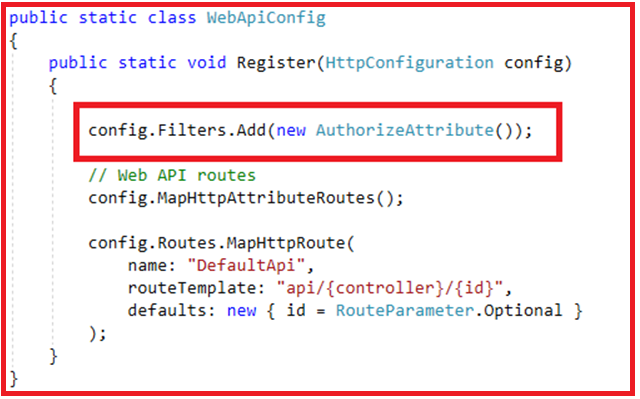
**Using the [Authorize] Attribute**

The ASP.NET Web API Framework provides a built-in authorization filter attribute i.e. **AuthorizeAttribute** and you can use this built-in filter attribute to checks whether the user is authenticated or not. If not, then it simply returns the HTTP status code **401 Unauthorized**, without invoking the controller action method.

You can apply the above built-in filter globally, at the controller level, or at the action level.

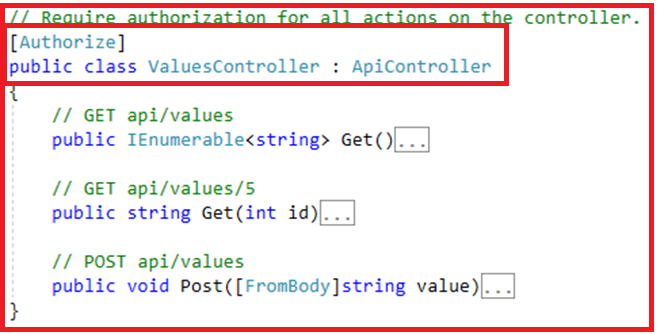
**At Globally**:

If you want to check the authentication for all the Web API controllers, then it is better to add the **AuthorizeAttribute** filter to the global filter list within the **Register** method of the **WebApiConfig** class as shown in the below image:



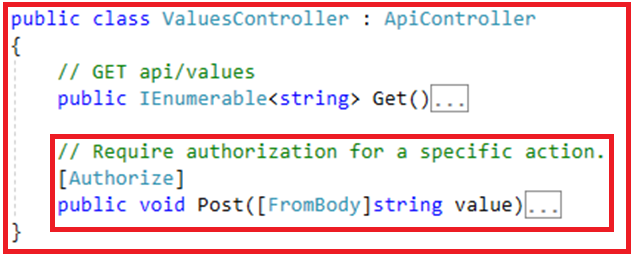
**At Controller Level**:

If you want to provide authentication for all the action methods of a specific controller, then it is better and recommended to add the **Authorize** filter at the controller level as shown in the below image.

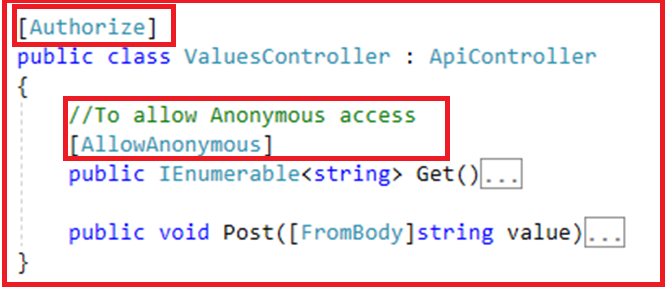


**At Action Level**:

If you want to provide authentication for specific action methods of a controller, then it is better to add the **Authorize**filter attribute to the action method which required authentication as shown in the below image.

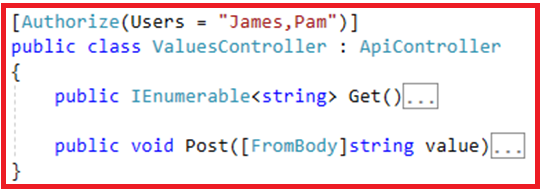


Another way of doing this is, restrict the controller by decorating the controller with **Authorize** filter attribute and then allow anonymous access to the action methods which does not require authentication by using the **AllowAnonymous** attribute. In the below example, the Post method is restricted, but the Get method allows anonymous access.

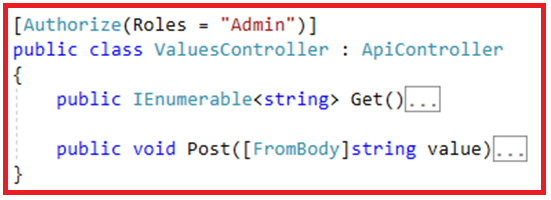


As of now, we have discussed two things. If we want to check the authentication before invoking the action method then we need to use the built-in **Authorize** Filter Attribute. If we want any action method to be accessed by the anonymous users then we need to decorate that action method with the **AllowAnonymous** attribute. Along the way, we can also limit access to specific users or to users with specific roles.

**Restrict by Users:**



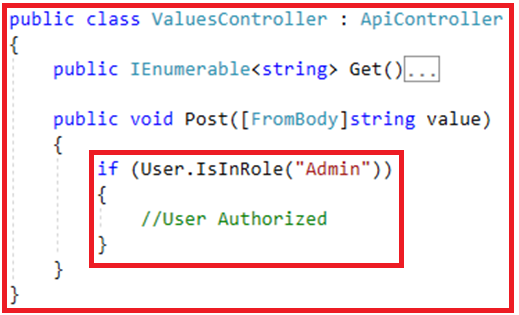
**Restrict by Roles:**



**Note:** The point to remember here is that the **AuthorizeAttribute**filter for Web API is located in the **System.Web.Http** namespace. In MVC there is also an **AuthorizeAttribute**filter which is located in the **System.Web.Mvc** namespace, which is not compatible with Web API controllers.

**Authorization Inside a Controller Action**

In some scenarios, you might allow a request to proceed, but you need to change the behavior based on the principal. For example, the information that you are going to return from the action depends on the user’s role. Within a controller action method, you can get the current principal object from the **ApiController.User** property is shown in the below image.



**ASP.NET Web API Basic Authentication**

In this article, I am going to discuss how to implement the **ASP.NET Web API Basic Authentication** step by step with an example. Please read our previous article where we discussed the basics of **[Authentication and Authorization in Web API.](https://dotnettutorials.net/lesson/authentication-and-authorization-in-web-api/)** As part of this article, we are going to discuss the following pointers.

1. **Why do we need Authentication in Web API?**
2. **How does Basic Authentication Work in Web API?**
3. **How to Implement Basic Authentication in ASP.NET Web API?**
4. **How to Enable Basic Authentication in Web API?**
5. **Testing the ASP.NET Web API Basic Authentication using Postman**

**Why do we need Authentication in Web API?**

Let’s start the discussion with one of the Rest Constraint i.e. Stateless Constraint. The Stateless Constraint is one of the Rest Constraints which states that the communication between the client and server must be stateless between the requests. This means that we should not be storing the client information on the server which required to process the request. The request that is coming from the client should contain all the necessary information that is required by the server to process that request. This ensures that each request coming from the client can be treated independently by the server.

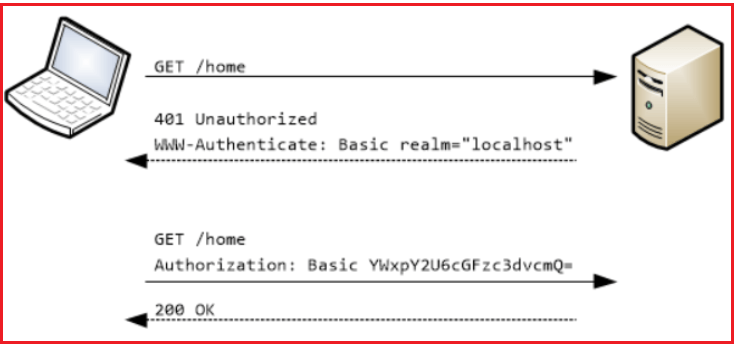
The above approach is fine and the advantage is that we can separate the client or server at any given point in time without affecting others. Here, the client can be any type of application including JavaScript or any other programming language like Java, PHP, or C#. The server does not remember the client once the request has been processed, So, each and every request coming from the client is new to the server and the server needs to check the request (most of the time the HTTP header) to identify the user.

So, in order to process the request by the server, the client needs to pass its credentials with each and every request and then the server will check and match the credentials with any persistent storage (most of the time it may be a database). If the credentials are found in the persistent storage then the server will treat that HTTP request as a valid request and process it else it simply returns an unauthorized error to the client.

We can implement Authentication and Authorization in many ways in an application. Here, in this article, I am going to discuss how to implement **ASP.NET Web API Basic Authentication**.

**How does Basic Authentication Work in Web API?**

Before implementing the Basic Authentication in ASP.NET Web API, let us first understand how does the basic authentication work in Web API? To understand how does basic authentication works, please have a look at the following diagram.



In Basic Authentication, if the client didn’t send the credentials in the request header (most of the time it is Authorization header), then the server will return**401** (**Unauthorized**). The response will also include a **WWW-Authenticate** header, indicating that the server supports Basic Authentication and that you can see in the above image for the first request which does not include the Authorization header.

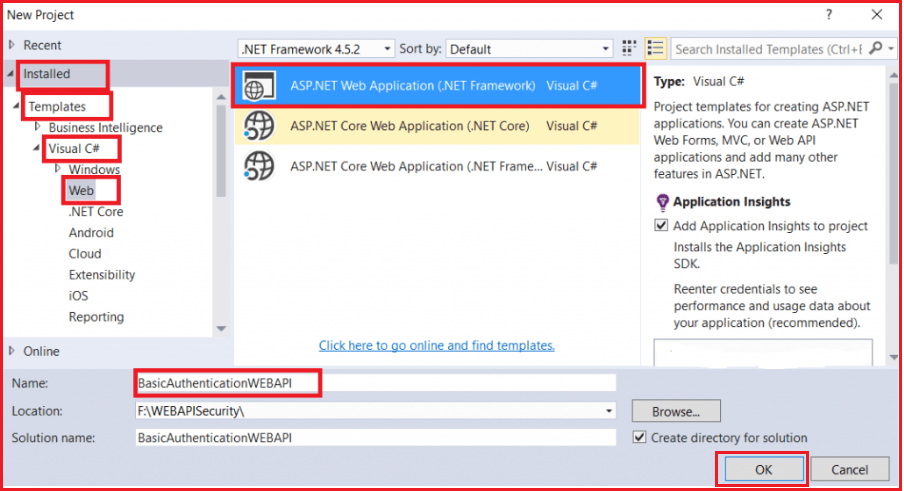
The client sends another request to the server, with the client credentials in the Authorization header. Generally, the client credentials are formatted as the string “**name:password**“, base64-encoded format and this time server validates the client and processes the request and if everything is fine, then you will get 200 OK status which you can see in the above image for the second request.

In Basic Authentication, as we attach the sensitive data (i,e. username and password) in each and every HTTP request, it should be transferred in an encoded format (base64-encoded format) and the protocol should be HTTPS, then only we can protect our data over the internet.

The **ASP.NET Web API Basic Authentication** is performed within the context of a “**realm**.” The server includes the name of the realm in the **WWW-Authenticate** header. The user’s credentials are valid within that realm. The exact scope of a realm is defined by the server. For example, you might define several realms in order to partition resources.

**Implementing Basic Authentication in ASP.NET Web API**

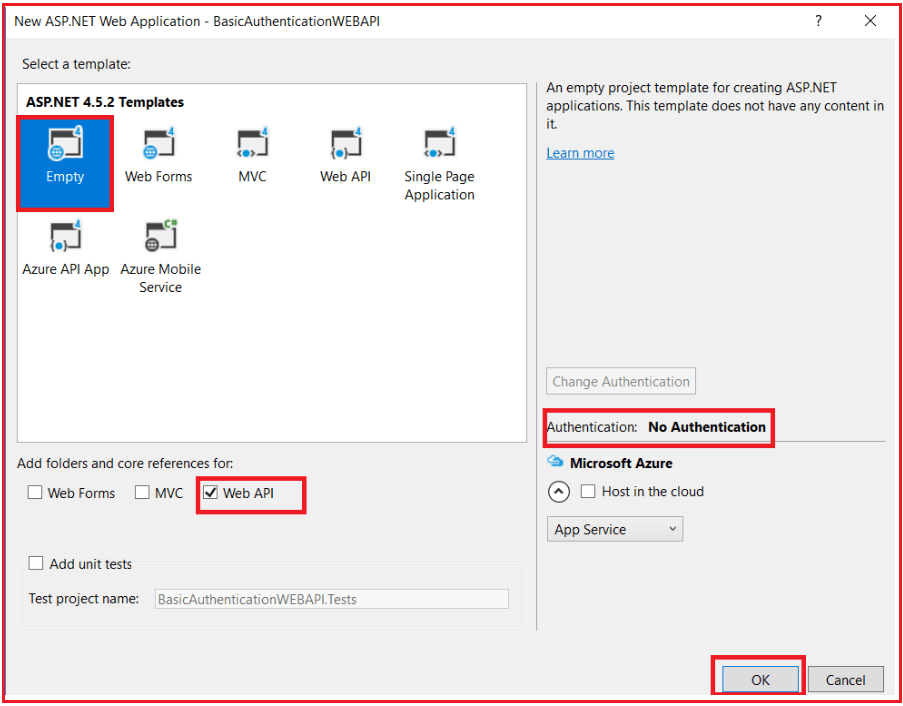
First, create an ASP.NET Web Application with the name **BasicAuthenticationWEBAPI** (you can give any name) as shown in the below image.



Once you click on the **OK** button, it will open the “**Select a template**” window. From the “**Select a template**” window choose

1. **Empty template**
2. **Web API Checkbox**
3. **No Authentication**

And finally, click on the **OK** button as shown below



Once you click on the **OK** Button it will take some time to create the project for us.

**Creating Models**

Now we need to create two models i.e. **User** and **Employee**. So Right-click on the Models folder and add a class file with the Name **User.cs** and then copy and paste the below code into it. This is a very simple class having only three properties i.e. ID, UserName and Password.

**namespace** *BasicAuthenticationWEBAPI.Models*

**{**

**public** **class** User

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string UserName **{** **get**; **set**; **}**

**public** string Password **{** **get**; **set**; **}**

**}**

**}**

Similarly, right-click on the Models folder and add a class file with the Name **Employee.cs** and then copy and paste the below code into it. This is also a very simple class having 5 properties i.e. ID, Name, Gender, Dept, and Salary.

**namespace** *BasicAuthenticationWEBAPI.Models*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string Dept **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**Creating Business Layer:**

Now we will create two classes that will return the list of users and the list of employees. Right-click on the Models folder and add a class file with the Name **UserBL.cs** and then copy and paste the below code. As you can see, here we created one method to return the list of users. In real-time, you will get the list of users from a database, but here, we are hardcoded the user’s list.

**namespace** *BasicAuthenticationWEBAPI.Models*

**{**

**public** **class** UsersBL

**{**

**public** List**<**User**>** GetUsers**()**

**{**

// In Real-time you need to get the data from any persistent storage

// For Simplicity of this demo and to keep focus on Basic Authentication

// Here we are hardcoded the data

List**<**User**>** userList = new List**<**User**>()**;

userList.Add**(**new User**()**

**{**

ID = 101,

UserName = "MaleUser",

Password = "123456"

**})**;

userList.Add**(**new User**()**

**{**

ID = 101,

UserName = "FemaleUser",

Password = "abcdef"

**})**;

**return** userList;

**}**

**}**

**}**

Similarly, right-click on the Models folder and add a class file with the Name **EmployeeBL.cs** and then copy and paste the below code into it. As you can see, here we created one method to return the list of employees. In real-time, you will get the list of employees from a database, but here, we are hardcoded the employee’s list.

**namespace** *BasicAuthenticationWEBAPI.Models*

**{**

**public** **class** EmployeeBL

**{**

**public** List**<**Employee**>** GetEmployees**()**

**{**

// In Real-time you need to get the data from any persistent storage

// For Simplicity of this demo and to keep focus on Basic Authentication

// Here we hardcoded the data

List**<**Employee**>** empList = new List**<**Employee**>()**;

**for** **(int** i = 0; i **<** 10; i++**)**

**{**

**if** **(**i **>** 5**)**

**{**

empList.Add**(**new Employee**()**

**{**

ID = i,

Name = "Name" + i,

Dept = "IT",

Salary = 1000 + i,

Gender = "Male"

**})**;

**}**

**else**

**{**

empList.Add**(**new Employee**()**

**{**

ID = i,

Name = "Name" + i,

Dept = "HR",

Salary = 1000 + i,

Gender = "Female"

**})**;

**}**

**}**

**return** empList;

**}**

**}**

**}**

Now, we need to create a class that will check whether the username and password are valid or not. Right-click on the Models folder and add a class file with the Name **UserValidate.cs** and then copy and paste the following code into it. As you can see, here, the Login method takes the username and password as input parameters. Then it will check whether the username and password are valid or not. If valid, then it returns TRUE indicating the user is valid else returns FALSE indicating the user is invalid.

**namespace** *BasicAuthenticationWEBAPI.Models*

**{**

**public** **class** UserValidate

**{**

//This method is used to check the user credentials

**public** **static** **bool** Login**(**string username, string password**)**

**{**

UsersBL userBL = new UsersBL**()**;

var UserLists = userBL.GetUsers**()**;

**return** UserLists.Any**(**user =**>**

user.UserName.Equals**(**username, StringComparison.OrdinalIgnoreCase**)**

&& user.Password == password**)**;

**}**

**}**

**}**

**Create a Basic Authentication Filter in ASP.NET Web API**

Right Click on the Models folder and add a class file with the name **BasicAuthenticationAttribute** and then copy and paste the following code in it. Here, the **BasicAuthenticationAttribute** class is inherited from the **AuthorizationFilterAttribute** class and overrides the OnAuthorization method which makes this class an **AuthorizationFilter**and can be applied like other attributes to the action methods or at the Controller level. Here, first, we are checking the Authorization header and if it is null, we are simply returning an Unauthorized error to the client. If the Authorization header is not null, then we are taking the Authorization header value, then we decode the value and then we split the decoded value and get the user name and password. Then we call the Login method of the UserValidate class to check if the user is a valid user or not. If the user is not valid, then we return an Unauthorized error to the client else we will proceed with the request.

**using** *System;*

**using** *System.Net;*

**using** *System.Net.Http;*

**using** *System.Security.Principal;*

**using** *System.Text;*

**using** *System.Threading;*

**using** *System.Web;*

**using** *System.Web.Http.Controllers;*

**using** *System.Web.Http.Filters;*

**namespace** *BasicAuthenticationWEBAPI.Models*

**{**

**public** **class** BasicAuthenticationAttribute : AuthorizationFilterAttribute

**{**

**private** const string Realm = "My Realm";

**public** **override** **void** OnAuthorization**(**HttpActionContext actionContext**)**

**{**

//If the Authorization header is empty or null

//then return Unauthorized

**if** **(**actionContext.Request.Headers.Authorization == **null)**

**{**

actionContext.Response = actionContext.Request

.CreateResponse**(**HttpStatusCode.Unauthorized**)**;

// If the request was unauthorized, add the WWW-Authenticate header

// to the response which indicates that it require basic authentication

**if** **(**actionContext.Response.StatusCode == HttpStatusCode.Unauthorized**)**

**{**

actionContext.Response.Headers.Add**(**"WWW-Authenticate",

string.Format**(**"Basic realm=\"{0}\"", Realm**))**;

**}**

**}**

**else**

**{**

//Get the authentication token from the request header

string authenticationToken = actionContext.Request.Headers

.Authorization.Parameter;

//Decode the string

string decodedAuthenticationToken = Encoding.UTF8.GetString**(**

Convert.FromBase64String**(**authenticationToken**))**;

//Convert the string into an string array

string**[]** usernamePasswordArray = decodedAuthenticationToken.Split**(**':'**)**;

//First element of the array is the username

string username = usernamePasswordArray**[**0**]**;

//Second element of the array is the password

string password = usernamePasswordArray**[**1**]**;

//call the login method to check the username and password

**if** **(**UserValidate.Login**(**username, password**))**

**{**

var identity = new GenericIdentity**(**username**)**;

IPrincipal principal = new GenericPrincipal**(**identity,**null)**;

Thread.CurrentPrincipal = principal;

**if** **(**HttpContext.Current != **null)**

**{**

HttpContext.Current.User = principal;

**}**

**}**

**else**

**{**

actionContext.Response = actionContext.Request

.CreateResponse**(**HttpStatusCode.Unauthorized**)**;

**}**

**}**

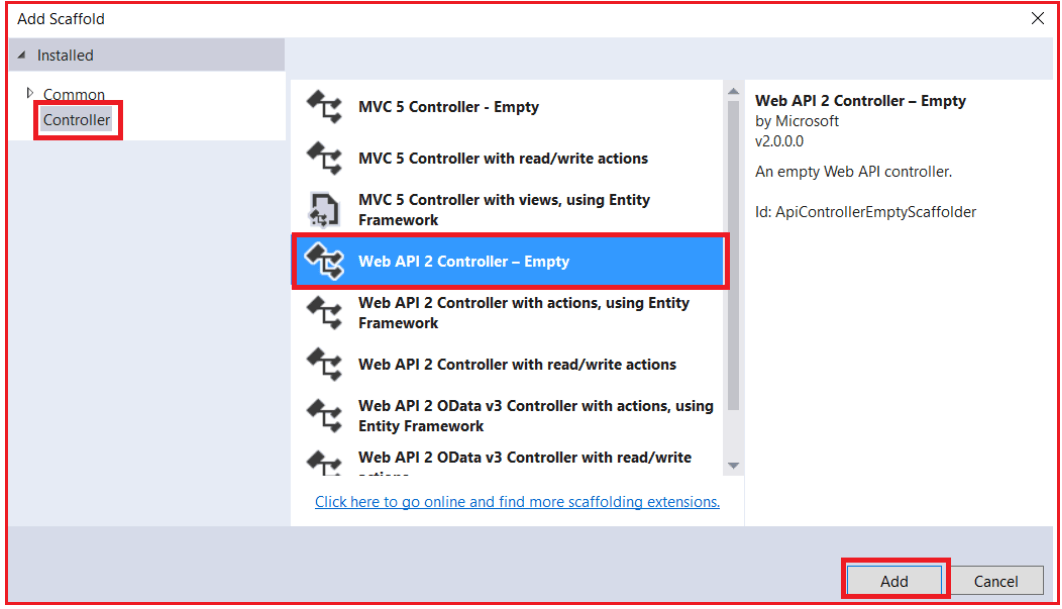
**}**

**}**

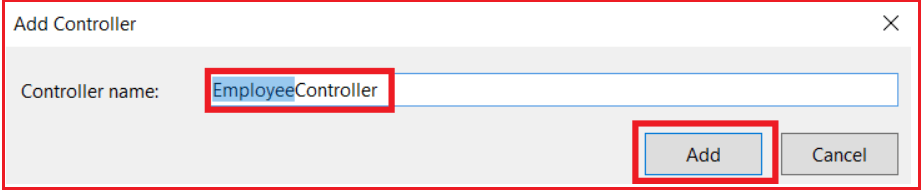
**}**

**Adding WebAPI2 Empty Controller**

Right-click on the Controllers folder and select **Add => Controller** which will open the window to select the controller as shown below.



From this window select **Web API 2 Controller – Empty** and click on the **Add** button, which will open another window to give a name to your controller as shown below.

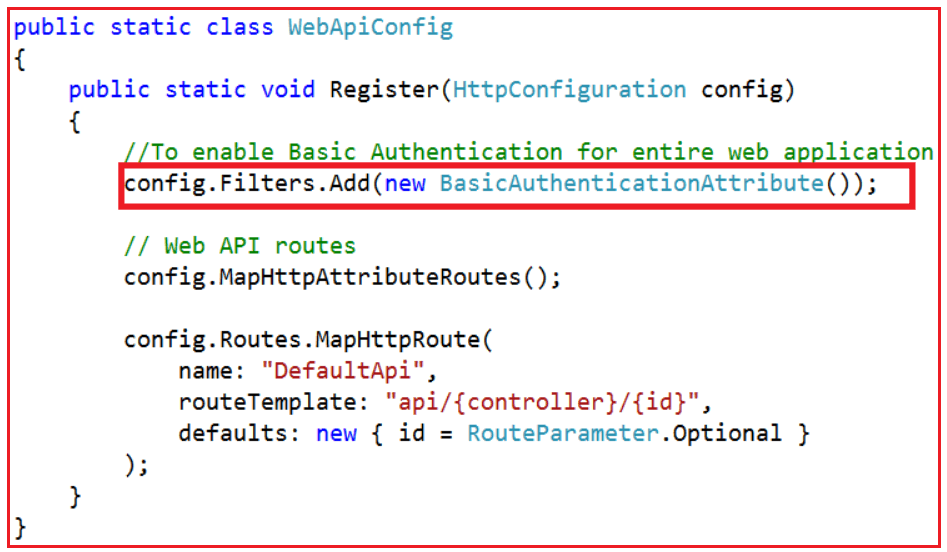


Provide the controller name as Employee and click on the Add button which will add Employee Controller within the controller folder.

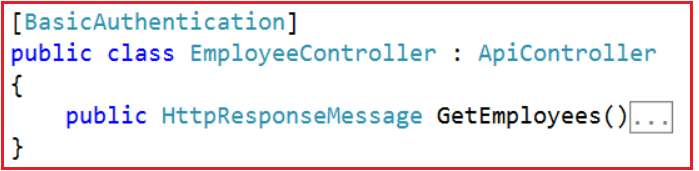
**Enable Web API Basic Authentication**

We can enable basic authentication in many different ways by applying the **BasicAuthenticationAttribute**. We can apply the **BasicAuthenticationAttribute** attribute on a specific controller, specific action, or globally which will be applicable to all Web API controllers and action methods.

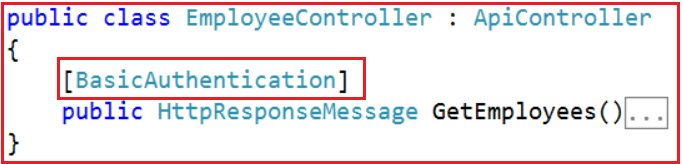
To enable the basic authentication across the entire ASP.NET Web API application, register the **BasicAuthenticationAttribute** as a filter using the **Register()** method in **WebApiConfig** class as shown in the below image.



We can also apply the BasicAuthenticationAttribute attribute on a specific controller which will enable the basic authentication for all the methods that are present in that controller as shown in the below image.



You can also enable the basic authentication at the action method level as shown in the below image which is only applicable to that particular action method which is decorated with the BasicAuthenticationAttribute.



Let’s first add an action method to the Employee Controller with the following business requirements. As we have two users i.e. MaleUser and FemaleUser and if the user login with the MaleUser username we want to display all the “male” employees and if the user login with the FemaleUser username we want to display all the female employees. Along with the above business requirement, we also enable basic authentication at the action method level.

**Add the following action method within the Employee controller**

**namespace** *BasicAuthenticationWEBAPI.Controllers*

**{**

**public** **class** EmployeeController : ApiController

**{**

**[**BasicAuthentication**]**

**public** HttpResponseMessage GetEmployees**()**

**{**

string username = Thread.CurrentPrincipal.Identity.Name;

var EmpList = new EmployeeBL**()**.GetEmployees**()**;

**switch** **(**username.ToLower**())**

**{**

**case** "maleuser":

**return** Request.CreateResponse**(**HttpStatusCode.OK,

EmpList.Where**(**e =**>** e.Gender.ToLower**()** == "male"**)**.ToList**())**;

**case** "femaleuser":

**return** Request.CreateResponse**(**HttpStatusCode.OK,

EmpList.Where**(**e =**>** e.Gender.ToLower**()** == "female"**)**.ToList**())**;

**default**:

**return** Request.CreateResponse**(**HttpStatusCode.BadRequest**)**;

**}**

**}**

**}**

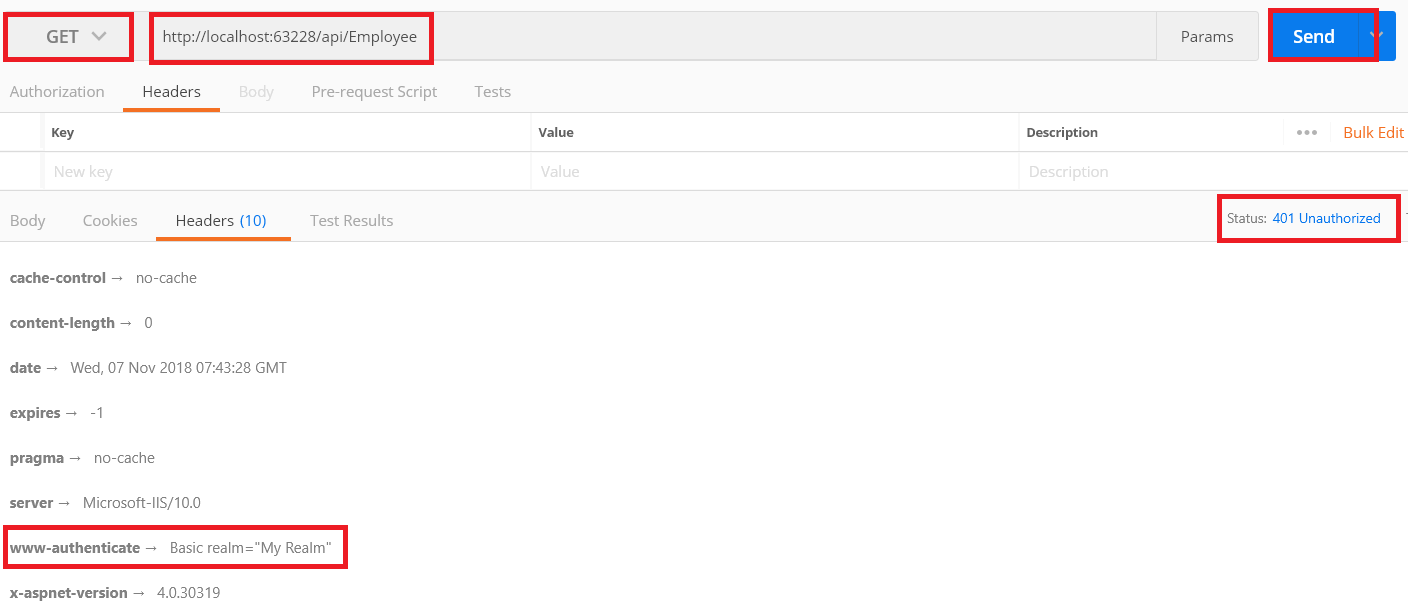
**}**

**Testing the Web API Basic Authentication using Postman**

If you are new to the postman, I strongly recommended you to read the following article, where I discussed how to download and use postman to test rest services.

**<https://dotnettutorials.net/lesson/how-to-use-postman-to-test-web-api/>**

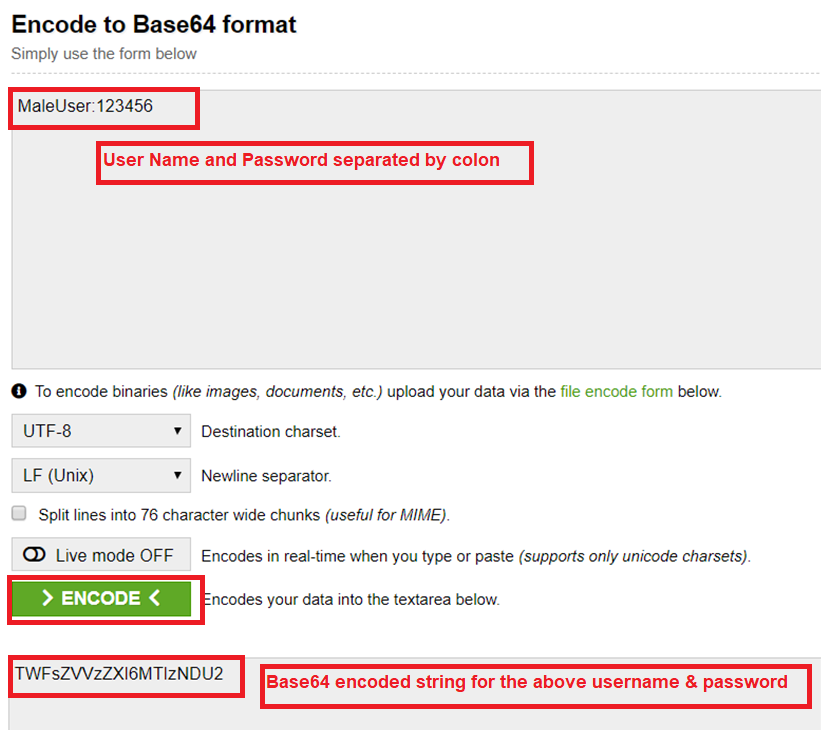
Let’s first make a request without passing the authorization header. Set the method type as GET, provide the requ**Web API Versioning**est URI and click on the Send button as shown in the below image.



Here you can observe that you will get a 401 status code which is Unauthorized. Let’s make the request to use the Authorization header. The username and password need to be a colon (:) separated and must be in base64 encoded. To do so, just use the following website

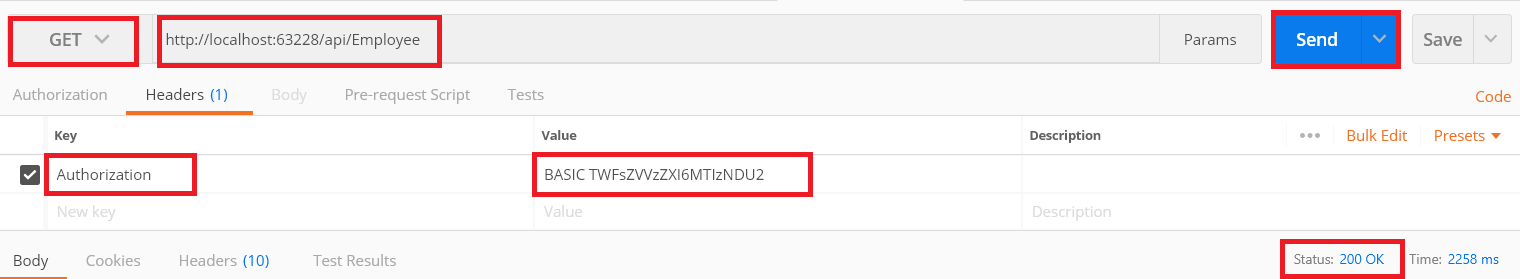
**<https://www.base64encode.org/>**

Enter the username and password separated by a colon (:) in the **“Encode to Base64 format”** textbox, and then click on the **“Encode”**button as shown in the below diagram which will generate the Base64 encoded value.



Once you generate the Base64 encoded string, let’s see how to use basic authentication in the header to pass the Base64 encoded value. Here we need to use the Authorization header and the value will be the Base64 encoded string followed the “BASIC” as shown below.

**Authorization: BASIC TWFsZVVzZXI6MTIzNDU2**



Once you click on the Send button, you can see that the status code is 200 as expected.

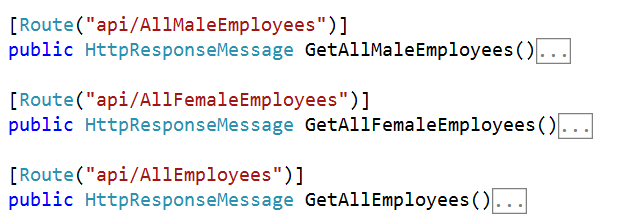
**Role-Based Basic Authentication in Web API**

In this article, I am going to discuss how to implement the **Role-Based Basic Authentication in ASP.NET Web API** Applications. Please read our last article before proceeding to this article, where we discussed **[How to implement ASP.NET Web API Basic Authentication](https://dotnettutorials.net/lesson/web-api-basic-authentication/)** with an example. As part of this article, we are going to discuss the following pointers related to authentication and authorization.

1. **Why do we need Role-Based Authentication?**
2. **How to Implement Role-Based Basic Authentication in Web API?**
3. **Testing the Role-Based Basic Authentication using Postman.**
4. **What are the advantages and disadvantages of using BASIC Authentication in Web API?**

**Why do we need Role-Based Authentication?**

Let us understand this with an example. As shown in the below image, we have three resources i.e. **GetAllMaleEmployees**, **GetAllFemaleEmployees**, and **GetAllEmployees** in our service.



In our application, let say we have two types of **Roles** i.e. **Admin** and **Superadmin** and as per our business requirement,

1. Only the users who have the Role Admin can access only to the GetAllMaleEmployees resource.
2. The users who have the Role Superadmin can access only the GetAllFemaleEmployees resource.
3. The GetAllEmployees resource can be accessed by both the Admin and Superadmin resource.

In order to achieve this, we need to implement Role-Based Authentication in ASP.NET Web API.

**Implementing Role-Based Basic Authentication in Web API.**

First, create an empty Web API application with the name **RoleBasedBasicAuthenticationWEBAPI.**Then Add the following User and Employee model to the Models folder

**User.cs**

**namespace** *RoleBasedBasicAuthenticationWEBAPI.Models*

**{**

**public** **class** User

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string UserName **{** **get**; **set**; **}**

**public** string Password **{** **get**; **set**; **}**

**public** string Roles **{** **get**; **set**; **}**

**public** string Email **{** **get**; **set**; **}**

**}**

**}**

**Employee.cs**

**namespace** *RoleBasedBasicAuthenticationWEBAPI.Models*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string Dept **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

Now we need to add the UserBL and EmployeeBL class file within the Models folder.

**UserBL.cs**

**namespace** *RoleBasedBasicAuthenticationWEBAPI.Models*

**{**

**public** **class** UsersBL

**{**

**public** List**<**User**>** GetUsers**()**

**{**

// In Realtime you need to get the data from any persistent storage

// For Simplicity of this demo and to keep focus on Basic Authentication

// Here we are hardcoded the data

List**<**User**>** userList = new List**<**User**>()**;

userList.Add**(**new User**()**

**{**

ID = 101,

UserName = "AdminUser",

Password = "123456",

Roles = "Admin",

Email = "Admin@a.com"

**})**;

userList.Add**(**new User**()**

**{**

ID = 102,

UserName = "BothUser",

Password = "abcdef",

Roles = "Admin,Superadmin",

Email = "BothUser@a.com"

**})**;

userList.Add**(**new User**()**

**{**

ID = 103,

UserName = "SuperadminUser",

Password = "Password@123",

Roles = "Superadmin",

Email = "Superadmin@a.com"

**})**;

**return** userList;

**}**

**}**

**}**

**EmployeeBL.cs**

**namespace** *RoleBasedBasicAuthenticationWEBAPI.Models*

**{**

**public** **class** EmployeeBL

**{**

**public** List**<**Employee**>** GetEmployees**()**

**{**

// In Realtime you need to get the data from any persistent storage

// For Simplicity of this demo and to keep focus on Basic Authentication

// Here we hardcoded the data

List**<**Employee**>** empList = new List**<**Employee**>()**;

**for** **(int** i = 0; i **<** 10; i++**)**

**{**

**if** **(**i **>** 5**)**

**{**

empList.Add**(**new Employee**()**

**{**

ID = i,

Name = "Name" + i,

Dept = "IT",

Salary = 1000 + i,

Gender = "Male"

**})**;

**}**

**else**

**{**

empList.Add**(**new Employee**()**

**{**

ID = i,

Name = "Name" + i,

Dept = "HR",

Salary = 1000 + i,

Gender = "Female"

**})**;

**}**

**}**

**return** empList;

**}**

**}**

**}**

Now add one more class file with the name UserValidate and copy and paste the following code.

**UserValidate.cs**

**namespace** *RoleBasedBasicAuthenticationWEBAPI.Models*

**{**

**public** **class** UserValidate

**{**

//This method is used to check the user credentials

**public** **static** **bool** Login**(**string username, string password**)**

**{**

UsersBL userBL = new UsersBL**()**;

var UserLists = userBL.GetUsers**()**;

**return** UserLists.Any**(**user =**>**

user.UserName.Equals**(**username, StringComparison.OrdinalIgnoreCase**)**

&& user.Password == password**)**;

**}**

//This method is used to return the User Details

**public** **static** User GetUserDetails**(**string username, string password**)**

**{**

UsersBL userBL = new UsersBL**()**;

**return** userBL.GetUsers**()**.FirstOrDefault**(**user =**>**

user.UserName.Equals**(**username, StringComparison.OrdinalIgnoreCase**)**

&& user.Password == password**)**;

**}**

**}**

**}**

Now create the **BasicAuthenticationAttribute** which will implement the **AuthorizationFilterAttribute** where we will put the logic for role-based basic authentication.

**BasicAuthenticationAttribute.cs**

**using** *System;*

**using** *System.Net;*

**using** *System.Net.Http;*

**using** *System.Security.Claims;*

**using** *System.Security.Principal;*

**using** *System.Text;*

**using** *System.Threading;*

**using** *System.Web;*

**using** *System.Web.Http.Controllers;*

**using** *System.Web.Http.Filters;*

**namespace** *RoleBasedBasicAuthenticationWEBAPI.Models*

**{**

**public** **class** BasicAuthenticationAttribute : AuthorizationFilterAttribute

**{**

**private** const string Realm = "My Realm";

**public** **override** **void** OnAuthorization**(**HttpActionContext actionContext**)**

**{**

**if** **(**actionContext.Request.Headers.Authorization == **null)**

**{**

actionContext.Response = actionContext.Request.CreateResponse**(**HttpStatusCode.Unauthorized**)**;

**if** **(**actionContext.Response.StatusCode == HttpStatusCode.Unauthorized**)**

**{**

actionContext.Response.Headers.Add**(**"WWW-Authenticate", string.Format**(**"Basic realm=\"{0}\"", Realm**))**;

**}**

**}**

**else**

**{**

string authenticationToken = actionContext.Request.Headers.Authorization.Parameter;

string decodedAuthenticationToken = Encoding.UTF8.GetString**(**Convert.FromBase64String**(**authenticationToken**))**;

string**[]** usernamePasswordArray = decodedAuthenticationToken.Split**(**':'**)**;

string username = usernamePasswordArray**[**0**]**;

string password = usernamePasswordArray**[**1**]**;

**if** **(**UserValidate.Login**(**username, password**))**

**{**

var UserDetails = UserValidate.GetUserDetails**(**username, password**)**;

var identity = new GenericIdentity**(**username**)**;

identity.AddClaim**(**new Claim**(**"Email", UserDetails.Email**))**;

identity.AddClaim**(**new Claim**(**ClaimTypes.Name, UserDetails.UserName**))**;

identity.AddClaim**(**new Claim**(**"ID", Convert.ToString**(**UserDetails.ID**)))**;

IPrincipal principal = new GenericPrincipal**(**identity, UserDetails.Roles.Split**(**','**))**;

Thread.CurrentPrincipal = principal;

**if** **(**HttpContext.Current != **null)**

**{**

HttpContext.Current.User = principal;

**}**

**}**

**else**

**{**

actionContext.Response = actionContext.Request

.CreateResponse**(**HttpStatusCode.Unauthorized**)**;

**}**

**}**

**}**

**}**

**}<**b**>**

**<**/b**>**

Now we will create our custom Authorize Attribute which will inherit from **AuthorizeAttribute** where we will implement the logic to return an appropriate response when the Authorization failed.

**MyAuthorizeAttribute.cs**

**namespace** *RoleBasedBasicAuthenticationWEBAPI.Models*

**{**

**public** **class** MyAuthorizeAttribute : System.Web.Http.AuthorizeAttribute

**{**

// 401 (Unauthorized) - indicates that the request has not been applied because it lacks valid

// authentication credentials for the target resource.

// 403 (Forbidden) - when the user is authenticated but isn’t authorized to perform the requested

// operation on the given resource.

**protected** **override** **void** HandleUnauthorizedRequest**(**System.Web.Http.Controllers.HttpActionContext actionContext**)**

**{**

**if** **(**!HttpContext.Current.User.Identity.IsAuthenticated**)**

**{**

**base**.HandleUnauthorizedRequest**(**actionContext**)**;

**}**

**else**

**{**

actionContext.Response = new System.Net.Http.HttpResponseMessage**(**System.Net.HttpStatusCode.Forbidden**)**;

**}**

**}**

**}**

**}**

**Let’s create a Web API 2 Empty Controller with the name EmployeeController and copy and paste the following code.**

**using** *RoleBasedBasicAuthenticationWEBAPI.Models;*

**using** *System.Linq;*

**using** *System.Net;*

**using** *System.Net.Http;*

**using** *System.Security.Claims;*

**using** *System.Web.Http;*

**namespace** *RoleBasedBasicAuthenticationWEBAPI.Controllers*

**{**

**public** **class** EmployeeController : ApiController

**{**

**[**BasicAuthentication**]**

**[**MyAuthorize**(**Roles = "Admin"**)]**

**[**Route**(**"api/AllMaleEmployees"**)]**

**public** HttpResponseMessage GetAllMaleEmployees**()**

**{**

var identity = **(**ClaimsIdentity**)**User.Identity;

//Getting the ID value

var ID = identity.Claims

.FirstOrDefault**(**c =**>** c.Type == "ID"**)**.Value;

//Getting the Email value

var Email = identity.Claims

.FirstOrDefault**(**c =**>** c.Type == "Email"**)**.Value;

//Getting the Username value

var username = identity.Name;

//Getting the Roles only if you set the roles in the claims

//var Roles = identity.Claims

// .Where(c => c.Type == ClaimTypes.Role)

// .Select(c => c.Value).ToArray();

var EmpList = new EmployeeBL**()**.GetEmployees**()**.Where**(**e =**>** e.Gender.ToLower**()** == "male"**)**.ToList**()**;

**return** Request.CreateResponse**(**HttpStatusCode.OK, EmpList**)**;

**}**

**[**BasicAuthentication**]**

**[**MyAuthorize**(**Roles = "Superadmin"**)]**

**[**Route**(**"api/AllFemaleEmployees"**)]**

**public** HttpResponseMessage GetAllFemaleEmployees**()**

**{**

var EmpList = new EmployeeBL**()**.GetEmployees**()**.Where**(**e =**>** e.Gender.ToLower**()** == "female"**)**.ToList**()**;

**return** Request.CreateResponse**(**HttpStatusCode.OK, EmpList**)**;

**}**

**[**BasicAuthentication**]**

**[**MyAuthorize**(**Roles = "Admin,Superadmin"**)]**

**[**Route**(**"api/AllEmployees"**)]**

**public** HttpResponseMessage GetAllEmployees**()**

**{**

var EmpList = new EmployeeBL**()**.GetEmployees**()**;

**return** Request.CreateResponse**(**HttpStatusCode.OK, EmpList**)**;

**}**

**}**

**}**

That’s it. We have done with our implementation.

**Testing Role-Based Basic Authentication in Web API using Postman**

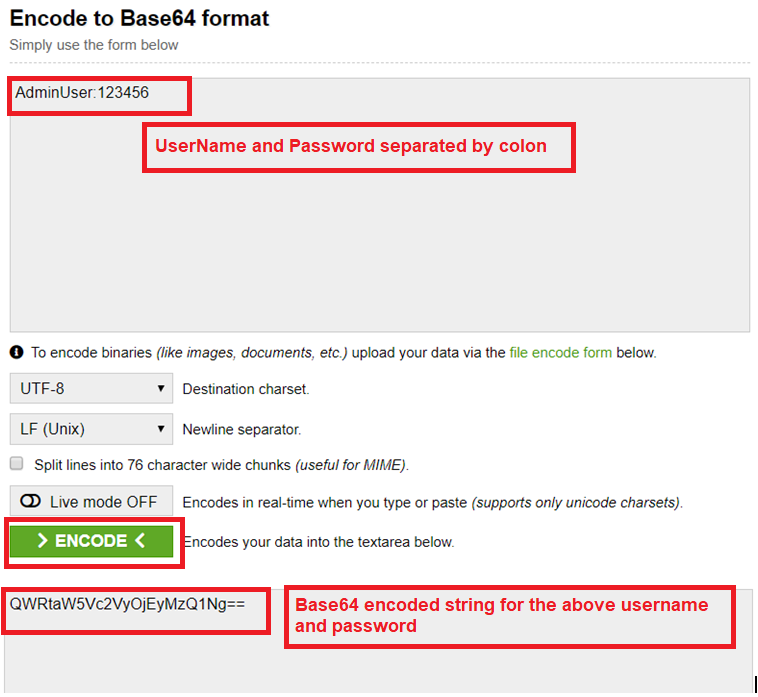
If you are new to the postman, I strongly recommended you to read the following article, where I discussed how to download and use postman to test rest services.

<https://dotnettutorials.net/lesson/how-to-use-postman-to-test-web-api/>

We need to pass the username and password in the Authorization header. The username and password need to be a colon (:) separated and must be in base64 encoded. To do so, just use the following website

<https://www.base64encode.org/>

Enter the username and password separated by a colon (:) in the **“Encode to Base64 format”** textbox, and then click on the **“Encode”**button as shown in the below diagram which will generate the Base64 encoded value. Let first generate the Base64 encoded string for the user AdminUser as shown in the below image



Once you generated the Base64 encoded string, let’s see how to use basic authentication in the header to pass the Base64 encoded value. Here we need to use the Authorization header and the value will be the Base64 encoded string followed the “BASIC” as shown below.

**Authorization: BASIC TWFsZVVzZXI6MTIzNDU2**

The role Admin has been assigned to the AdminUser. So he can access only the following two resources

**/api/AllMaleEmployees**  
**/api/AllEmployees**

But he cannot access the following resource

**/api/AllFemaleEmployees**

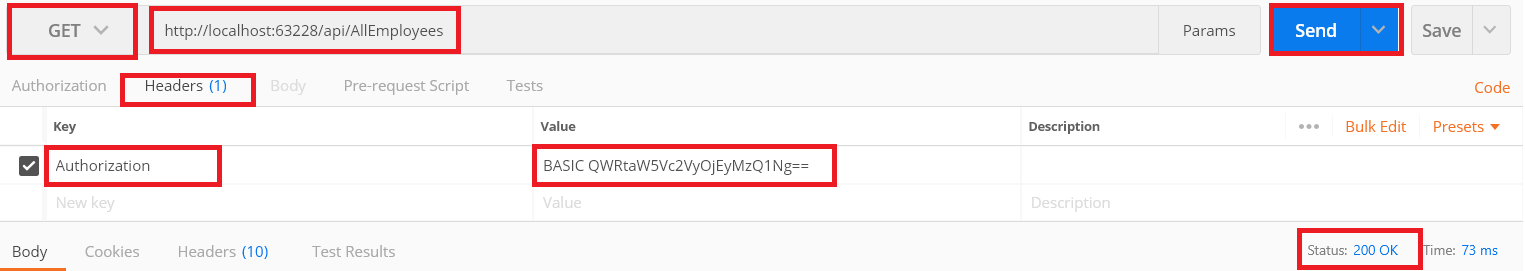
**Let proofs this using the Postman.**

**/api/AllMaleEmployees**



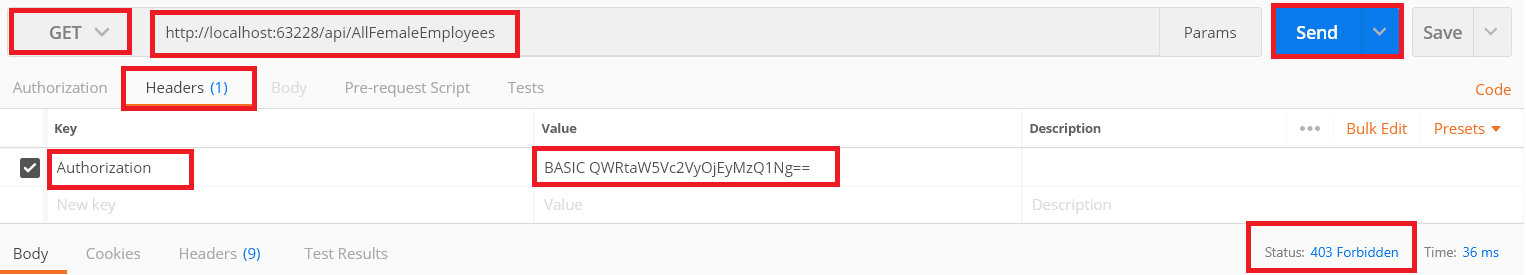
**Here we got the response 200 OK.**

**/api/AllEmployees**



**Here we also got the response 200 OK as expected.**

**/api/AllFemaleEmployees**



As you can see, here we got the response as **403 Forbidden** which means the user is authenticated but not authorized to access the above resource. Similarly, you can test the other users.

**Advantages and disadvantages of Basic Authentication in Web API.**

**Advantages:**

1. Internet standard.
2. Supported by all major browsers.
3. Relatively simple protocol.

**Disadvantages:**

1. User credentials are sent in the request.
2. Credentials are sent as plaintext.
3. Credentials are sent with every request.
4. No way to log out, except by ending the browser session.
5. Vulnerable to cross-site request forgery (CSRF); requires anti-CSRF measures.

**Web API Service with Basic Authentication**

In this article, I am going to discuss **how to consume Web API Services with Basic Authentication**. Please read our previous article before proceeding to this article, where we discussed how to implement the **[Role-Based Basic Authentication in Web API](https://dotnettutorials.net/lesson/role-based-web-api-authentication/)** with an example.  We are also going to work with our previous example. As part of this article, we are going to discuss the following pointers.

1. **How to enable CORS in Web API?**
2. **How to call Web API service from JQuery using Ajax?**
3. **How to call Web API service from C# using a Console Application?**

**How to Enable CORS in Web API?**

As we are going to consume the Web API Service using Jquery Ajax from another domain, we need to enable CORS in our application. Enabling CORS is a two steps process.

**Step1:**Install **Microsoft.AspNet.WebApi.Cors** package. Execute the following command using the NuGet Package Manager Console.

**Step2:** Include the following 2 lines of code in **Register()**method of **WebApiConfig**class in **WebApiConfig.cs** file which is inside **App\_Start**folder

**EnableCorsAttribute cors = new EnableCorsAttribute(“\*”, “\*”, “\*”);**  
**config.EnableCors();**

**With the above two lines of code in place, the WebApiConfig class should look as below.**

**using** *System.Web.Http;*

**using** *System.Web.Http.Cors;*

**namespace** *RoleBasedBasicAuthenticationWEBAPI*

**{**

**public** **static** **class** WebApiConfig

**{**

**public** **static** **void** Register**(**HttpConfiguration config**)**

**{**

// Web API routes

config.MapHttpAttributeRoutes**()**;

config.Routes.MapHttpRoute**(**

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new **{** id = RouteParameter.Optional **}**

**)**;

EnableCorsAttribute cors = new EnableCorsAttribute**(**"\*", "\*", "\*"**)**;

config.EnableCors**()**;

**}**

**}**

**}**

If you are new to CORS, then I strongly recommended you to read the following articles, where I discussed CORS in details.

**<https://dotnettutorials.net/lesson/cross-origin-resource-sharing-web-api/>**

**Modifying the Controller:**

In the controller, we need to create one method called GetEmployees. Depending on the credentials provided the Web API service should authenticate and return the correct results as follows.

If the AdminUser username and password are provided then only the male employees are returned from the service. Similarly, if SuperadminUser username and password are provided then only the female employees are returned from the service. In the same way, if both user username and password are provided then all the employees should be returned from the service else any other case it should return BAD Request from the service

**To achieve this, let’s modify the Employee Controller as shown below.**

**using** *RoleBasedBasicAuthenticationWEBAPI.Models;*

**using** *System.Linq;*

**using** *System.Net;*

**using** *System.Net.Http;*

**using** *System.Security.Claims;*

**using** *System.Threading;*

**using** *System.Web.Http;*

**namespace** *RoleBasedBasicAuthenticationWEBAPI.Controllers*

**{**

**public** **class** EmployeeController : ApiController

**{**

**[**BasicAuthentication**]**

**[**EnableCorsAttribute**(**"\*", "\*", "\*"**)]**

**[**MyAuthorize**(**Roles = "Admin,Superadmin"**)]**

**[**Route**(**"api/Employees"**)]**

**public** HttpResponseMessage GetEmployees**()**

**{**

//var identity = (ClaimsIdentity)User.Identity;

//var username = identity.Name;

//OR you can use the below code to get the login username

string username = Thread.CurrentPrincipal.Identity.Name;

var EmpList = new EmployeeBL**()**.GetEmployees**()**;

**switch** **(**username.ToLower**())**

**{**

**case** "adminuser":

**return** Request.CreateResponse**(**HttpStatusCode.OK,

EmpList.Where**(**e =**>** e.Gender.ToLower**()** == "male"**)**.ToList**())**;

**case** "superadminuser":

**return** Request.CreateResponse**(**HttpStatusCode.OK,

EmpList.Where**(**e =**>** e.Gender.ToLower**()** == "female"**)**.ToList**())**;

**case** "bothuser":

**return** Request.CreateResponse**(**HttpStatusCode.OK,EmpList**)**;

**default**:

**return** Request.CreateResponse**(**HttpStatusCode.BadRequest**)**;

**}**

**}**

**}**

**}**

In the last article, we discussed how to consume the Web APIs using Postman as a client. But it is also necessary to know how to consume APIs from different types of clients.

**Consuming Web API Service using JQuery AJAX:**

Create one empty web application and then add one HTML page with the name AJAXClient to your application. Once you add the HTML Page then copy and paste the following code.

**Note:** You need to be installed JQuery packages in your projects.

<!DOCTYPE html>

**<html>**

**<head>**

**<title></title>**

**<meta** charset="utf-8" **/>**

<!--First Installed Jquery into your application-->

**<script** src="Scripts/jquery-3.3.1.js"**></script>**

**<script** type="text/javascript"**>**

$(document).ready(function () {

var ulEmployees = $('#ulEmployees');

$('#btnGetData').click(function () {

// Get the username & password from textboxes

var username = $('#txtUsername').val();

var password = $('#txtPassword').val();

$.ajax({

type: 'GET',

// Make sure to change the port number to

// where you have the service

// running on your local machine

url: 'http://localhost:63228/api/Employees',

dataType: 'json',

// Specify the authentication header

// btoa() method encodes a string to Base64

headers: {

'Authorization': 'Basic ' + btoa(username + ':' + password)

},

success: function (data) {

ulEmployees.empty();

$.each(data, function (index, val) {

var EmployeeDetails = 'Name = '+ val.Name + ' Gender = ' + val.Gender + ' Dept = ' + val.Dept + ' Salary = ' + val.Salary;

ulEmployees.append('**<li>**' + EmployeeDetails + '**</li>**')

});

},

complete: function (jqXHR) {

if (jqXHR.status == '401') {

ulEmployees.empty();

ulEmployees.append('**<li** style="color:red"**>**'

+ jqXHR.status + ' : ' + jqXHR.statusText + '**</li>**')

}

}

});

});

$('#btnClear').click(function () {

ulEmployees.empty();

});

});

**</script>**

**</head>**

**<body>**

Username : **<input** type="text" id="txtUsername" **/>**

Password : **<input** type="password" id="txtPassword" **/>**

**<br** **/><br** **/>**

**<input** id="btnGetData" type="button" value="Get Employees" **/>**

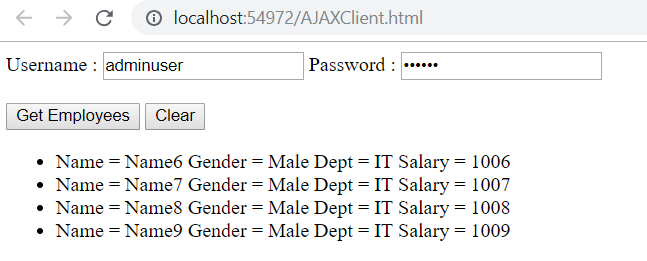
**<input** id="btnClear" type="button" value="Clear" **/>**

**<ul** id="ulEmployees"**></ul>**

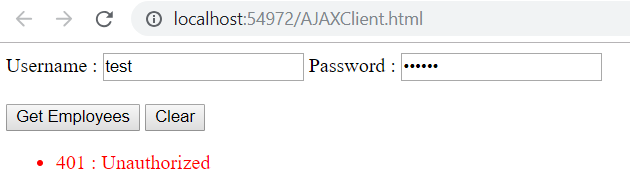
**</body>**

**</html>**

First, run the service application and then the client application. Now go to the client application and navigate to the URL **/AJAXClient.html.**Provide the valid username and password and click on the GetEmployees button and you should see the data as expected.



Now provide the wrong username and password and see the results as 401 unauthorized as expected.



**Consuming Web API Service using C# Console Application:**

Create a console application and then copy and paste the following code. You need to Install **Newtonsoft.Json** Package into your application

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Text;*

**using** *System.Net.Http;*

**using** *System.Net.Http.Headers;*

**using** *Newtonsoft.Json;*

**namespace** *ClientApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

HttpClientHandler handler = new HttpClientHandler**()**;

HttpClient client = new HttpClient**(**handler**)**;

client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue**(**"Authorization",

Convert.ToBase64String**(**Encoding.Default.GetBytes**(**"AdminUser:123456"**)))**;

//Need to change the PORT number where your WEB API service is running

var result = client.GetAsync**(**new Uri**(**"http://localhost:63228/api/Employees"**))**.Result;

**if** **(**result.IsSuccessStatusCode**)**

**{**

Console.WriteLine**(**"Done" + result.StatusCode**)**;

var JsonContent = result.Content.ReadAsStringAsync**()**.Result;

List**<**Employee**>** empList = JsonConvert.DeserializeObject**<**List**<**Employee**>>(**JsonContent**)**;

**foreach(**var emp in empList**)**

**{**

Console.WriteLine**(**"Name = " + emp.Name + " Gender = " + emp.Gender + " Dept = " + emp.Dept + " Salary = " + emp.Salary**)**;

**}**

**}**

**else**

Console.WriteLine**(**"Error" + result.StatusCode**)**;

Console.ReadLine**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

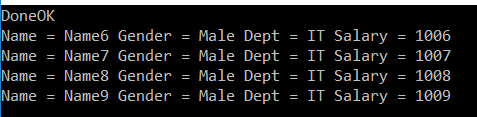
**public** string Dept **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

Here, we have given the AdminUser username and password, so it displays only the male employees as expected as shown in the below image.



**HTTP Message Handlers in WEB API**

**Server-Side HTTP Message Handlers in ASP.NET Web API**

In this article, I am going to discuss the **Server-Side HTTP Message Handlers in ASP.NET Web API** with Examples. Please read our previous article where we discussed **[How to Consume Web API Services](https://dotnettutorials.net/lesson/consuming-web-api-service-with-basic-authentication/)** with Basic Authentication. As part of this article, we are going to discuss the following important concepts related to Message handlers.

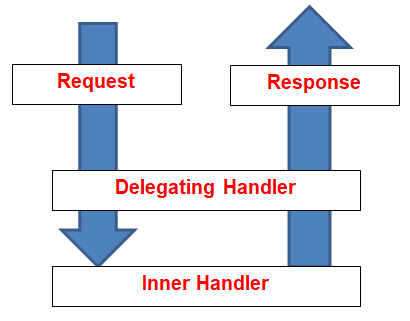
1. **What is an HTTP Message handler in ASP.NET Web API Application?**
2. **Understanding Delegating Handler in Web API.**
3. **Different Types of HTTP Message Handlers in Web API.**
4. **Understanding Server-Side HTTP Message Handlers in Web API,**
5. **How to Create Custom Server-Side HTTP Message Handlers in Web API?**
6. **How to add Custom HTTP Message Handlers to the Pipeline.**
7. **Different Types of Examples to understand the HTTP Message Handlers.**

**What is an HTTP Message handler in ASP.NET Web API Application?**

An **HTTP Message Handler in ASP.NET Web API** is a class that receives an HTTP request and returns an HTTP response. The **Message Handler** is derived from the abstract **HttpMessageHandler** class.

**Understanding Delegating Handler in Web API:**

To handle the HTTP Request and to generate the HTTP Response in ASP.NET Web API, a series of message handlers are chained together. The first handler in the chain receives the HTTP request. Do some processing, and gives the request to the next handler. At some point, the response is generated and goes back up in the chain. This pattern is called delegating handler. The following diagram shows this process.



**What are the Different Types of HTTP Message Handlers available in ASP.NET Web API?**

In ASP.NET Web API Framework, there are two types of message handlers are available. They are as follows

1. **Server-Side HTTP Message Handlers**
2. **Client-Side HTTP Message Handlers**

In this article, I am going to discuss the **Server-Side HTTP Message Handlers** in the Web API Framework with real-time examples. In our next article, I am going to discuss the **[Client-Side HTTP Message Handlers](https://dotnettutorials.net/lesson/http-client-message-handler/)** in ASP.NET Web API.

**Understanding Server-Side HTTP Message Handlers in Web API**

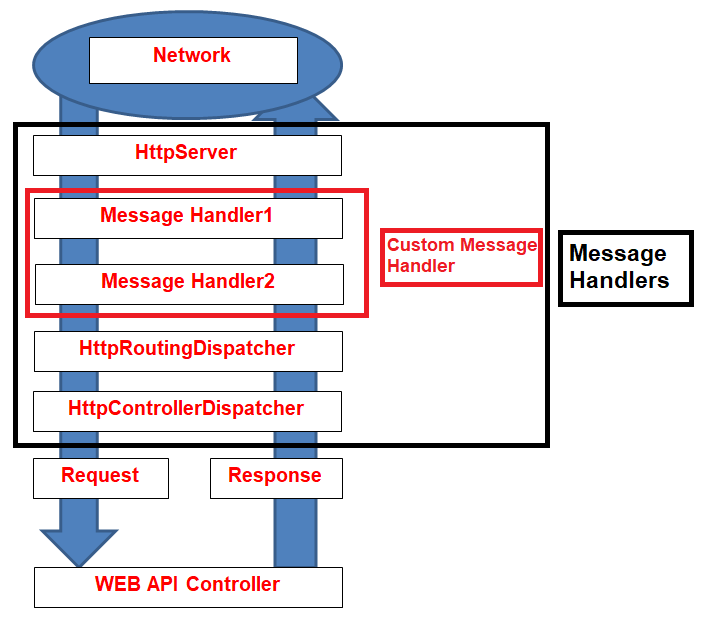
On the server-side, the ASP.NET Web API Framework uses some built-in message handlers which are as follows:

1. **HttpServer**: This built-in message handler gets the request from the host.
2. **HttpRoutingDispatcher**: This message handler dispatches the request based on the route.
3. **HttpControllerDispatcher**: This message handler sends the request to an ASP.NET Web API controller.

You can also create your own custom handlers and then add them to the Web API pipeline. The Message handlers are good for cross-cutting concerns (such as authentication and authorization) that operate at the level of HTTP messages rather than controller actions. For example, a custom message handler might do the following things

1. **Read or modify the HTTP request headers.**
2. **Add a response header to the HTTP response.**
3. **Validate the requests before they reach the controller (i.e. Authentication and Authorization).**

The following diagram shows two custom handlers (Message Handler1 and Message Handler2) inserted into the Web API pipeline at the Server side.



**Creating Custom Server-Side HTTP Message Handlers in Web API**

As we already discussed along with the built-in Server-side Message Handlers, you can also create your own Server-Side HTTP Message Handlers. So let’s discuss how to create the **Custom Server-Side HTTP Message handlers** in ASP.NET Web API.

To create a custom Server-Side HTTP Message Handler in ASP.NET Web API, you need to create a class that must be derived from the **System.Net.Http.DelegatingHandler.**That custom class then should override the **SendAsync** method. **SendAsync** method has the following signature:



The **SendAsync** method takes an **HttpRequestMessage** as input and asynchronously returns an **HttpResponseMessage**. A typical implementation does the following:

1. Process the request message.
2. Call the **base.SendAsync** method to send the request to the inner handler.
3. The inner handler returns a response message. (This step is asynchronous.)
4. Process the response message and returns the response to the caller.

**A sample code is shown below.**

**using** *System.Diagnostics;*

**using** *System.Net.Http;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *MessageHandler.Models*

**{**

**public** **class** MessageHandler1 : DelegatingHandler

**{**

**protected** **async** **override** Task**<**HttpResponseMessage**>** SendAsync**(**

HttpRequestMessage request, CancellationToken cancellationToken**)**

**{**

Debug.WriteLine**(**"Process request"**)**;

// Call the inner handler.

var response = **await** **base**.SendAsync**(**request, cancellationToken**)**;

Debug.WriteLine**(**"Process response"**)**;

**return** response;

**}**

**}**

**}**

**Note:**The call to the **base.SendAsync** is asynchronous. If the handler does any work after this call, use the await keyword, as shown in the above example.

A delegating handler can also skip the inner handler and directly create the response. Let’s look at the below example which exactly does the same thing.

**using** *System.Net;*

**using** *System.Net.Http;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *MessageHandler.Models*

**{**

**public** **class** MessageHandler2 : DelegatingHandler

**{**

**protected** **override** Task**<**HttpResponseMessage**>** SendAsync**(**

HttpRequestMessage request, CancellationToken cancellationToken**)**

**{**

// Create the response.

var response = new HttpResponseMessage**(**HttpStatusCode.OK**)**

**{**

Content = new StringContent**(**"Hello!"**)**

**}**;

// Note: TaskCompletionSource creates a task that does not contain a delegate.

var tsc = new TaskCompletionSource**<**HttpResponseMessage**>()**;

tsc.SetResult**(**response**)**;

**return** tsc.Task;

**}**

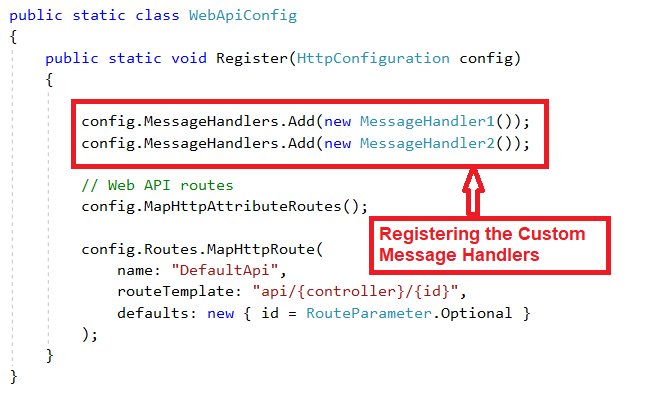
**}**

**}**

When a delegate handler creates the response without calling the **base.SendAsync** method, then the request skips the rest of the pipeline. This can be useful for a handler that validates the request creating an error response. We will discuss this with a complete example in our upcoming article.

**How to Add Custom HTTP Message Handlers to the Pipeline in ASP.NET Web API?**

To add the Custom HTTP Message Handlers on the server-side, you need to add the Custom HTTP Message Handlers to the **HttpConfiguration.MessageHandlers**collection inside the Register method of the **WebApiConfig** class as shown in the below image:



The Message Handlers are going to call in the same order as they appear in the **MessageHandlers** collection. The reason is they are nested and the response message travels in the other direction. That is, the last handler is the first one to get the response message.

Notice that you don’t need to set the inner handlers. That is the job of the Web API framework which will automatically connect the inner message handlers. Now let’s look at some of the examples of custom message handlers.

**Example: X-HTTP-Method-Override**

The **X-HTTP-Method-Override** is a non-standard HTTP header. It is basically designed for clients who cannot send certain HTTP request types, such as PUT or DELETE. Instead, the client sends a POST request and sets the **X-HTTP-Method-Override** header to the desired method type. For example **X-HTTP-Method-Override: PUT**

So to support this, we need to create a custom message handler that adds support for X-HTTP-Method-Override:

**using** *System;*

**using** *System.Linq;*

**using** *System.Net.Http;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *MessageHandler.Models*

**{**

**public** **class** XHTTPMethodOverrideHandler : DelegatingHandler

**{**

**readonly** string**[]** \_methods = **{** "DELETE", "HEAD", "PUT" **}**;

const string \_header = "X-HTTP-Method-Override";

**protected** **override** Task**<**HttpResponseMessage**>** SendAsync**(**

HttpRequestMessage request, CancellationToken cancellationToken**)**

**{**

// Check for HTTP POST with the X-HTTP-Method-Override header.

**if** **(**request.Method == HttpMethod.Post && request.Headers.Contains**(**\_header**))**

**{**

// Check if the header value is in our methods list.

var method = request.Headers.GetValues**(**\_header**)**.FirstOrDefault**()**;

**if** **(**\_methods.Contains**(**method, StringComparer.InvariantCultureIgnoreCase**))**

**{**

// Change the request method.

request.Method = new HttpMethod**(**method**)**;

**}**

**}**

**return** **base**.SendAsync**(**request, cancellationToken**)**;

**}**

**}**

**}**

**Explanation of the above code:**

In the above **SendAsync** method, the handler checks whether the request message is a POST request and whether it contains the **X-HTTP-Method-Override** header. If so, then it validates the request header value and then modifies the request method. Finally, the handler calls the base.SendAsync to pass the message to the next handler.

When the request reaches the **HttpControllerDispatcher** class, **HttpControllerDispatcher** will route the request based on the updated request method.

**Example: Adding a Custom Response Header**

Let’s see an example of a message handler that adds a custom header to every response message:

**using** *System.Net.Http;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *MessageHandler.Models*

**{**

**public** **class** CustomHeaderHandler : DelegatingHandler

**{**

**async** **protected** **override** Task**<**HttpResponseMessage**>** SendAsync**(**

HttpRequestMessage request, CancellationToken cancellationToken**)**

**{**

HttpResponseMessage response = **await** **base**.SendAsync**(**request, cancellationToken**)**;

response.Headers.Add**(**"X-Custom-Header", "This is my custom header."**)**;

**return** response;

**}**

**}**

**}**

In the above example, the handler first calls the **base.SendAsync** method to pass the request to the inner message handler. The inner handler returns a response message, but it does so asynchronously by using the **Task<T>** object. The response message is not available until the **base.SendAsync** completes asynchronously.

**Example: Checking for an API Key**

Some web services require the clients to include an API key in their request. The following example shows how a message handler can check the requests for a valid API key:

**using** *System.Net;*

**using** *System.Net.Http;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *MessageHandler.Models*

**{**

**public** **class** ApiKeyHandler : DelegatingHandler

**{**

**public** string Key **{** **get**; **set**; **}**

**public** ApiKeyHandler**(**string key**)**

**{**

this.Key = key;

**}**

**protected** **override** Task**<**HttpResponseMessage**>** SendAsync**(**

HttpRequestMessage request, CancellationToken cancellationToken**)**

**{**

**if** **(**!ValidateKey**(**request**))**

**{**

var response = new HttpResponseMessage**(**HttpStatusCode.Forbidden**)**;

var tsc = new TaskCompletionSource**<**HttpResponseMessage**>()**;

tsc.SetResult**(**response**)**;

**return** tsc.Task;

**}**

**return** **base**.SendAsync**(**request, cancellationToken**)**;

**}**

**private** **bool** ValidateKey**(**HttpRequestMessage message**)**

**{**

var query = message.RequestUri.ParseQueryString**()**;

string key = query**[**"key"**]**;

**return** **(**key == Key**)**;

**}**

**}**

**}**

The above message handler looks for the API key in the URI query string. (For the above example, we assume that the key is a static string. A real implementation would probably use more complex validation.) If the query string contains the key, the handler passes the request to the inner handler.

If the request does not have a valid key, the handler creates a response message with status 403, Forbidden. In this case, the handler does not call **base.SendAsync**, so the inner handler never receives the request, nor does the controller. Therefore, the controller can assume that all incoming requests have a valid API key.

**Note:** If the API key applies only to certain controller actions, consider using an action filter instead of a message handler. Action filters run after URI routing is performed.

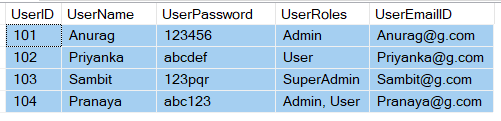
**Basic Authentication Using Message Handler in Web API**

In this article, I am going to discuss how to implement **Basic** **Authentication Using Message Handler**in ASP.NET Web API. Please read our last article, where I discussed the **[Server-Side HTTP Message Handler](https://dotnettutorials.net/lesson/http-message-handlers-in-web-api/)** in ASP.NET Web API.

As we already discussed, the basic authentication says that the client needs to send the username and password in base64 encoded format in the authorization header of the HTTP request. The server then gets the username and password from the authorization header. Once the username and password get from the header, then the server check and match the credentials with any persistent storage (most of the time it may be a database). If the credentials are found in the persistent storage then the server will treat that HTTP request as a valid request and process it else it simple return unauthorized response to the client.

**Let’s us implement Basic Authentication Using Message Handler**

We are going to use the following UserMaster table in this demo



Please use below SQL Script to create and populate the UserMaster table with the required sample data.

**CREATE** **DATABASE** SECURIRT\_DB

**GO**

**USE** [SECURIRT\_DB]

**CREATE** **TABLE** UserMaster

(

UserID **INT** **PRIMARY** **KEY**,

UserName **VARCHAR**(50),

UserPassword **VARCHAR**(50),

UserRoles **VARCHAR**(500),

UserEmailID **VARCHAR**(100),

)

**GO**

**INSERT** **INTO** UserMaster **VALUES**(101, 'Anurag', '123456', 'Admin', 'Anurag@g.com')

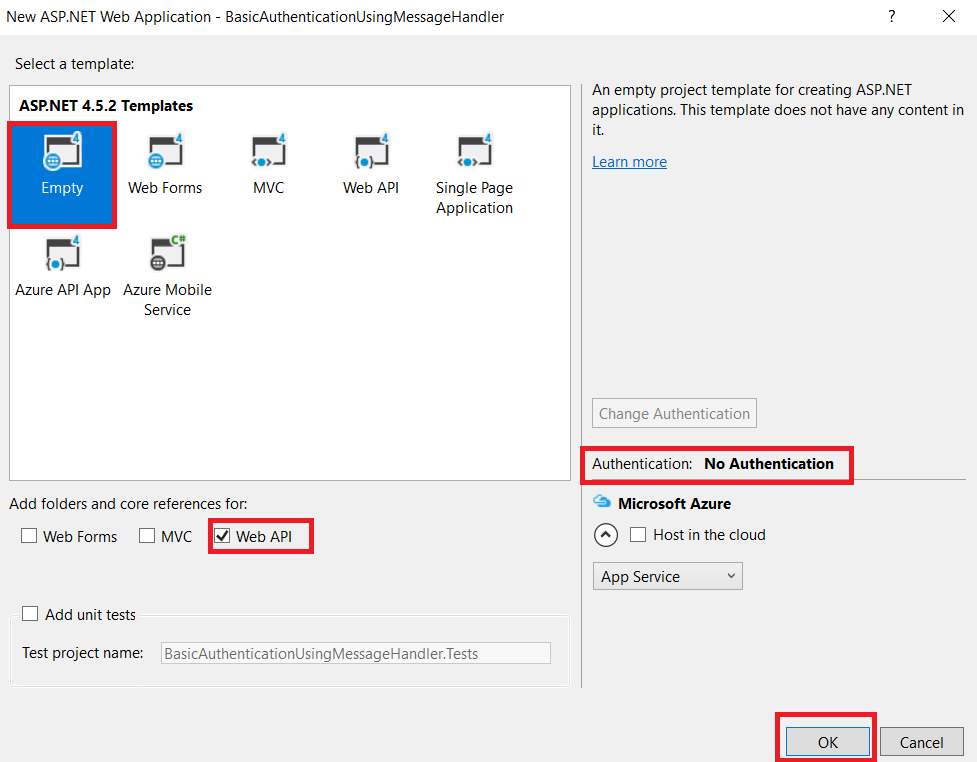
**INSERT** **INTO** UserMaster **VALUES**(102, 'Priyanka', 'abcdef', 'User', 'Priyanka@g.com')

**INSERT** **INTO** UserMaster **VALUES**(103, 'Sambit', '123pqr', 'SuperAdmin', 'Sambit@g.com')

**INSERT** **INTO** UserMaster **VALUES**(104, 'Pranaya', 'abc123', 'Admin, User', 'Pranaya@g.com')

**GO**

Let’s create an empty Web API application with the name BasicAuthenticationUsingMessageHandler (you can give any name) and select empty and Web API as shown in the below image.



Once you click on the OK button, it will create the application for us. Then the next step is to create an ADO.NET Entity Data Model against the SecurityDB and Select the UserMaster table.

Here you need to choose DB First Approach of Entity Framework.

**Let’s create a class with the name ValidateUser and copy and paste the following code.**

**namespace** *BasicAuthenticationUsingMessageHandler.Models*

**{**

**public** **class** ValidateUser

**{**

//This method is used to check the user credentials

**public** UserMaster CheckUserCredentials**(**string username, string password**)**

**{**

// SECURIRT\_DBEntities it is your context class

**using** **(**var context = new SECURIRT\_DBEntities**())**

**{**

**return** context.UserMasters.FirstOrDefault**(**user =**>**

user.UserName.Equals**(**username, StringComparison.OrdinalIgnoreCase**)**

&& user.UserPassword == password**)**;

**}**

**}**

**}**

**}**

In the above class, we create one method i.e. CheckUserCredentials which will validate the user by checking the username and password.

Now, let’s implement our own custom message handler to check whether or not the client has sent an Authorization header along with the HTTP request, if it is presented then we will check the header value against the persistent storage, in our case, it’s the database table.

So let’s create a class with the name BasicAuthenticationMessageHandler and copy and paste the following code.

**BasicAuthenticationMessageHandler.cs**

**namespace** *BasicAuthenticationUsingMessageHandler.Models*

**{**

**public** **class** BasicAuthenticationMessageHandler : DelegatingHandler

**{**

**protected** **override** Task**<**HttpResponseMessage**>** SendAsync**(**HttpRequestMessage request, CancellationToken cancellationToken**)**

**{**

**try**

**{**

var authenticationToken = request.Headers.GetValues**(**"Authorization"**)**.FirstOrDefault**()**;

**if** **(**authenticationToken != **null)**

**{**

**byte[]** data = Convert.FromBase64String**(**authenticationToken**)**;

string decodedAuthenticationToken = Encoding.UTF8.GetString**(**data**)**;

string**[]** UsernamePasswordArray = decodedAuthenticationToken.Split**(**':'**)**;

string username = UsernamePasswordArray**[**0**]**;

string password = UsernamePasswordArray**[**1**]**;

UserMaster ObjUser = new ValidateUser**()**.CheckUserCredentials**(**username, password**)**;

**if** **(**ObjUser != **null)**

**{**

var identity = new GenericIdentity**(**ObjUser.UserName**)**;

identity.AddClaim**(**new Claim**(**"Email", ObjUser.UserEmailID**))**;

IPrincipal principal = new GenericPrincipal**(**identity, ObjUser.UserRoles.Split**(**','**))**;

Thread.CurrentPrincipal = principal;

**if** **(**HttpContext.Current != **null)**

**{**

HttpContext.Current.User = principal;

**}**

**return** **base**.SendAsync**(**request, cancellationToken**)**;

**}**

**else**

**{**

var response = new HttpResponseMessage**(**HttpStatusCode.Unauthorized**)**;

var tsc = new TaskCompletionSource**<**HttpResponseMessage**>()**;

tsc.SetResult**(**response**)**;

**return** tsc.Task;

**}**

**}**

**else**

**{**

var response = new HttpResponseMessage**(**HttpStatusCode.BadRequest**)**;

var tsc = new TaskCompletionSource**<**HttpResponseMessage**>()**;

tsc.SetResult**(**response**)**;

**return** tsc.Task;

**}**

**}**

**catch**

**{**

var response = new HttpResponseMessage**(**HttpStatusCode.Forbidden**)**;

var tsc = new TaskCompletionSource**<**HttpResponseMessage**>()**;

tsc.SetResult**(**response**)**;

**return** tsc.Task;

**}**

**}**

**}**

**}**

**Please add the following namespaces:**

**using** *System;*

**using** *System.Linq;*

**using** *System.Net;*

**using** *System.Net.Http;*

**using** *System.Security.Claims;*

**using** *System.Security.Principal;*

**using** *System.Text;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

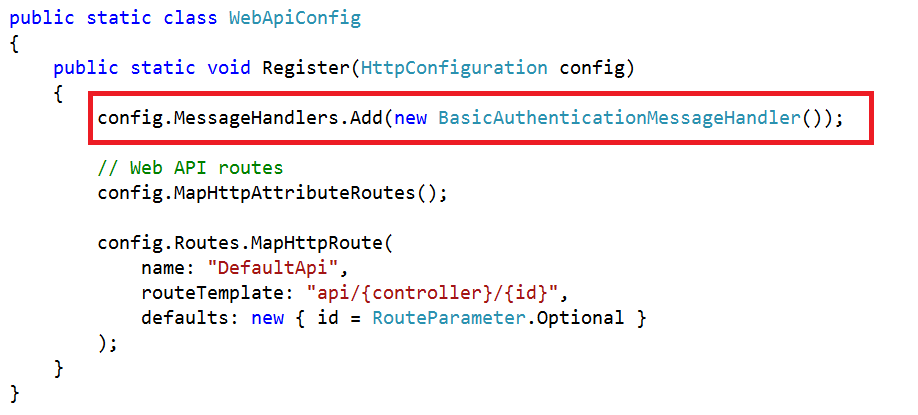
**using** *System.Web;*

**Explanation of the above code:**

If the Authorization header is not present in the HTTP request then it will be considered as a forbidden request but if it is present then we will get the header value. Once we get the header value then we need to decode as the value of the header is comes in encoded. Here we will use the Base64 encoding scheme in the attached header.

Once we get the user credentials then we will check the credentials and if the credentials are present in the database then we will consider it as a valid user and we will set the user principals along with the current thread.

The request will then be redirected towards a specific controller and action. Register the custom handler in the WebApiConfig file as shown in the below diagram.



That’s it. We are done with our implementation. Let’s create one empty Web API controller and then copy and paste the following code.

**using** *System.Net;*

**using** *System.Net.Http;*

**using** *System.Threading;*

**using** *System.Web.Http;*

**namespace** *BasicAuthenticationUsingMessageHandler.Controllers*

**{**

**public** **class** TestController : ApiController

**{**

**[**Authorize**(**Roles = "Admin,User"**)]**

**public** HttpResponseMessage Get**()**

**{**

//You can implement youe own logic

//Get the Identity Name

string username = Thread.CurrentPrincipal.Identity.Name;

**return** Request.CreateResponse**(**HttpStatusCode.OK, "User Name = "+ username**)**;

**}**

**[**Authorize**(**Roles = "Admin"**)]**

**public** HttpResponseMessage Post**()**

**{**

string username = Thread.CurrentPrincipal.Identity.Name;

**return** Request.CreateResponse**(**HttpStatusCode.OK, "User Name = " + username**)**;

**}**

**}**

**}**

As you can see in the above controller, both the Get and Post are decorated with an Authorise attribute and we have specified the role over each action. So, the specific roles can access a specific action. Since the Get() is to read the data, generally both the Admin and the User can access it but a Post is only allowed for an Admin.

**Testing using Postman**

First, let’s test for the following user

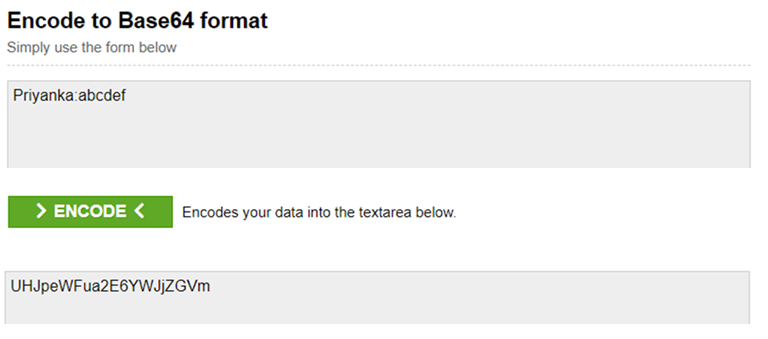
UserName: Priyanka

Password: abcdef

The username and password need to be a colon (:) separated and must be in base64 encoded. To do this use the following website

**<https://www.base64encode.org/>**

Enter the username and password separated by a colon (:) in **“Encode to Base64 format”** textbox, and then click on the **“Encode”**button as shown in the below diagram which will generate the Base64 encoded value.

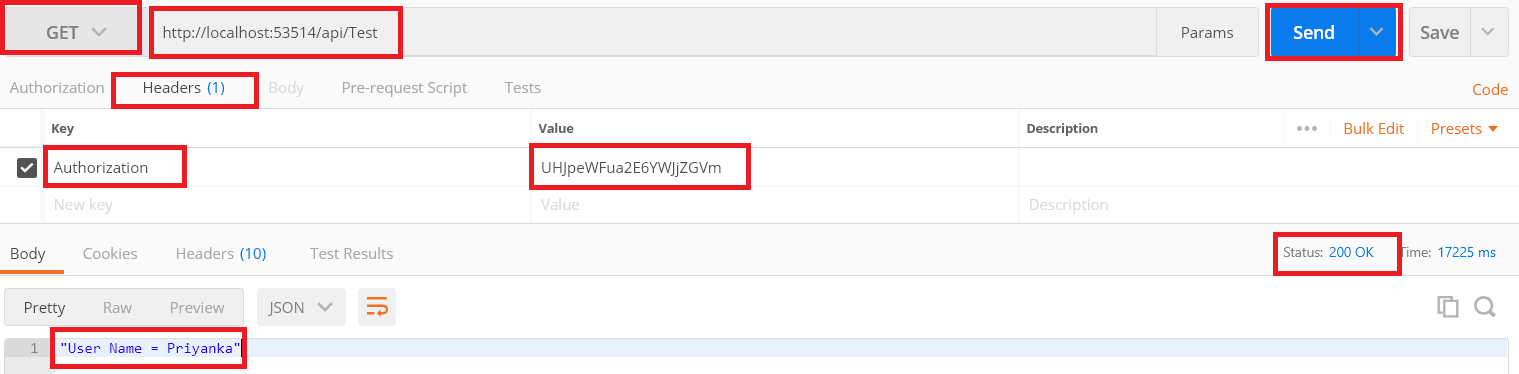


Once you generated the Base64 encoded string, let’s see how to use basic authentication in the header to pass the Base64 encoded value.

Here we need to use the Authorization header and the value will be the **Base64 encoded** string as shown below.

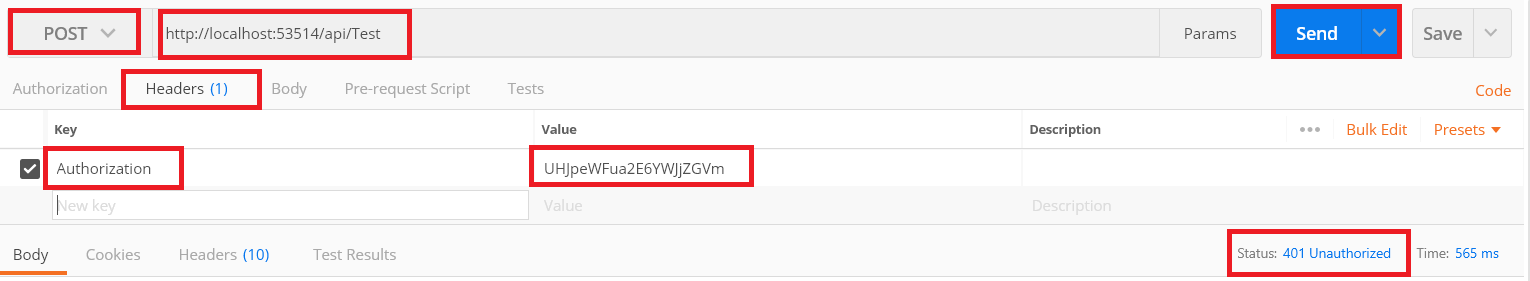
**Authorization: UHJpeWFua2E6YWJjZGVm**

**GET Request:**



As you can see, we get Status 200 as expected as the user Priyanka has the role “User” and the role “User” has access to the Get Method of the Test Controller.

**POST Request:**



As you can see, we get Status 401 Unauthorized as expected as the user Priyanka has the role “User” and the role “User” does not have access to the Post Method of the Test Controller.

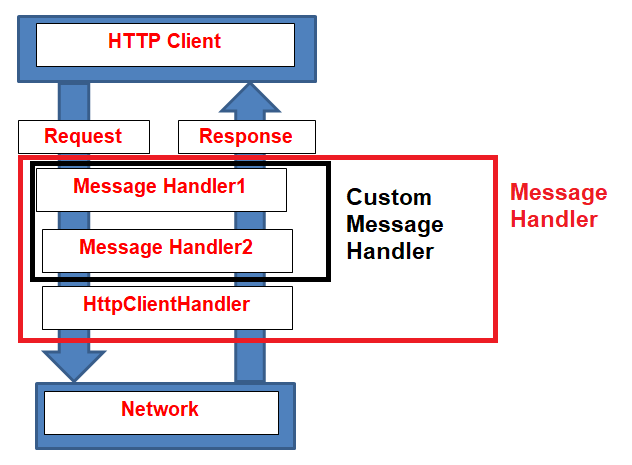
**HTTP Client Message Handler in Web API**

In this article, I am going to discuss **HTTP Client Message Handler in Web API**with real-time examples. As we already discussed in **[HTTP Message Handler Article](https://dotnettutorials.net/lesson/http-message-handlers-in-web-api/)** that a **Message Handler**is a class that receives an **HTTP request** and returns an **HTTP response**. The **Message handler** is derived from the abstract **HttpMessageHandler** class. There are two types of HTTP Message Handlers as follows

1. **[The Server Side HTTP Message Handlers](https://dotnettutorials.net/lesson/http-message-handlers-in-web-api/)**– we already discussed
2. Client Side HTTP Message Handlers – will discuss in this article

**HTTP Client Message Handlers in Web API**

The **HttpClient** class uses a message handler to process the requests on the client side. The default handler provided by the dot net framework is **HttpClientHandler.** This **HTTP Client Message Handler** sends the request over the network and also gets the response from the server. As a developer if you want, then you can also create your own custom message handlers and then insert the **custom message handlers** into the pipeline in the client side as shown in the below image.

****

**Creating a Custom HTTP Client Message Handler:**

Let us discuss how to create a Custom HTTP Client Message Handler. To create a custom HTTP Client message handler, what we need to do is, we need to create a custom class and that class should be derived from the **System.Net.Http.DelegatingHandler** class**.** Then the class should override the **SendAsync** method. The signature of the **SendAsync** method as following:



The **SendAsync** method takes an **HttpRequestMessage** as input and asynchronously returns an **HttpResponseMessage**. A typical implementation does the following:

1. Process the request message.
2. Call the base.SendAsync method to send the request to the inner handler.
3. The inner handler returns a response message. (This step is asynchronous.)
4. Process the response message and returns the response to the caller.

Creating a Custom Message Handler:

The following example shows the creation of a custom message handler which adds a custom header to the outgoing request:

**using** *System.Net.Http;*

**using** *System.Threading.Tasks;*

**namespace** *ClientSideMessageHandler.Models*

**{**

**class** MessageHandler1 : DelegatingHandler

**{**

**private** **int** \_count = 0;

**protected** **override** Task**<**HttpResponseMessage**>** SendAsync**(**

HttpRequestMessage request, System.Threading.CancellationToken cancellationToken**)**

**{**

System.Threading.Interlocked.Increment**(ref** \_count**)**;

request.Headers.Add**(**"X-Custom-Header", \_count.ToString**())**;

**return** **base**.SendAsync**(**request, cancellationToken**)**;

**}**

**}**

**}**

The call to the **base.SendAsync** method is asynchronous. If your handler going to do some work after this call, then use the **await** keyword to resume execution after the method completes.

The following example shows a handler that logs error codes. The example shows how to get at the response inside the handler.

**using** *System.IO;*

**using** *System.Net.Http;*

**using** *System.Threading.Tasks;*

**namespace** *ClientSideMessageHandler.Models*

**{**

**class** LoggingHandler : DelegatingHandler

**{**

StreamWriter \_writer;

**public** LoggingHandler**(**Stream stream**)**

**{**

\_writer = new StreamWriter**(**stream**)**;

**}**

**protected** **override** **async** Task**<**HttpResponseMessage**>** SendAsync**(**

HttpRequestMessage request, System.Threading.CancellationToken cancellationToken**)**

**{**

var response = **await** **base**.SendAsync**(**request, cancellationToken**)**;

**if** **(**!response.IsSuccessStatusCode**)**

**{**

\_writer.WriteLine**(**"{0}\t{1}\t{2}", request.RequestUri,

**(int)**response.StatusCode, response.Headers.Date**)**;

**}**

**return** response;

**}**

**protected** **override** **void** Dispose**(bool** disposing**)**

**{**

**if** **(**disposing**)**

**{**

\_writer.Dispose**()**;

**}**

**base**.Dispose**(**disposing**)**;

**}**

**}**

**}**

**Adding Message Handlers to the Client Pipeline**

To add a custom message handlers to **HttpClient**pipeline, we need to use the **HttpClientFactory.Create** method as shown below.

**HttpClient client = HttpClientFactory.Create(new MessageHandler1(), new MessageHandler2());**

The Message handlers are called in the order that we pass them into the **Create** method of the **HttpClientFactory** class. The reason is handlers are nested the response message travels in the other direction. That is, the last handler is the first to get the response **message.**

**Token Based Authentication in Web API**

In this article, I am going to discuss how to implement **Token Based Authentication in Web API** to secure the server resources with an example. Please read our previous article where we discussed how to implement **[Client-Side HTTP Message Handler](https://dotnettutorials.net/lesson/http-client-message-handler/)** with some examples. As part of this article, we are going to discuss the following pointers.

1. **Why do we need Token Based Authentication in Web API?**
2. **Advantages of using Token Based Authentication in ASP.NET Web API**
3. **How does the Token-Based Authentication work?**
4. **Implementing Token-Based Authentication in Web API.**
5. **Testing the Token Authentication using Postman.**

**Why do we need Token Based Authentication in Web API?**

The**ASP.NET Web API** is an ideal framework, provided by Microsoft that, to build **Web API’s**, i.e. **HTTP** based services on top of the .NET Framework. Once we develop the services using Web API then these services are going to be consumed by a broad range of clients, such as

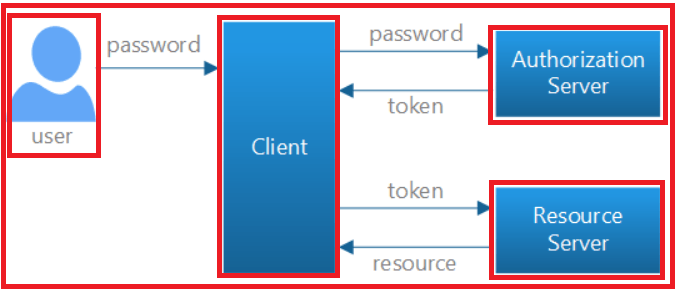
1. **Browsers**
2. **Mobile applications**
3. **Desktop applications**
4. **IOTs, etc.**

Nowadays, the use of WEB API is increasing in a rapid manner. So as a developer you should know how to develop Web APIs. Only developing Web APIs is not enough if there is no security. So, it also very important for us as a developer to implement security for all types of clients (such as Browsers, Mobile Devices, Desktop applications, and IoTs) who are going to use our Web API services.

The most preferred approach nowadays to secure the Web API resources is by authenticating the users in Web API server by using the **signed token** (which contains enough information to identify a particular user) which needs to be sent to the server by the client with each and every request. This is called the **Token-Based Authentication** approach.

**How does the Token-Based Authentication work?**

In order to understand how the token based authentication works, please have a look at the following diagram.



**The Token-Based Authentication works as Follows:**

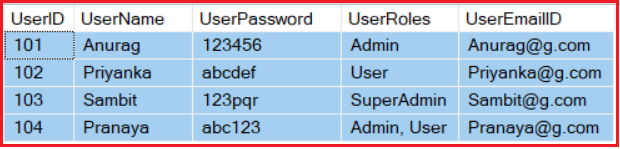
1. The user enters his credentials (i.e. the username and password) into the client (here client means the browser or mobile devices, etc).
2. The client then sends these credentials (i.e. username and password) to the Authorization Server.
3. Then the Authorization Server authenticates the client credentials (i.e. username and password) and generates and returns an access token. This Access Token contains enough information to identify a user and also contains the token expiry time.
4. The client application then includes the **Access Token in the Authorization header** of the HTTP request to access the restricted resources from the Resource Server until the token is expired.

Note: If this not clear at the moment then don’t worry, we will explain the above mentioned points one by one in detail with example.

**Let’s discuss the step by step procedure to implement Token-Based Authentication in Web API and then we will also how to use the token based authentication to access restricted resources using Postman and Fiddler.**

**Step1: Creating the required database**

We are going to use the following **UserMaster** table in this demo.



**Please use below SQL Script to create and populate the UserMaster table with the required sample data.**

**CREATE** **DATABASE** SECURITY\_DB

**GO**

**USE** [SECURITY\_DB]

**CREATE** **TABLE** UserMaster

(

UserID **INT** **PRIMARY** **KEY**,

UserName **VARCHAR**(50),

UserPassword **VARCHAR**(50),

UserRoles **VARCHAR**(500),

UserEmailID **VARCHAR**(100),

)

**GO**

**INSERT** **INTO** UserMaster **VALUES**(101, 'Anurag', '123456', 'Admin', 'Anurag@g.com')

**INSERT** **INTO** UserMaster **VALUES**(102, 'Priyanka', 'abcdef', 'User', 'Priyanka@g.com')

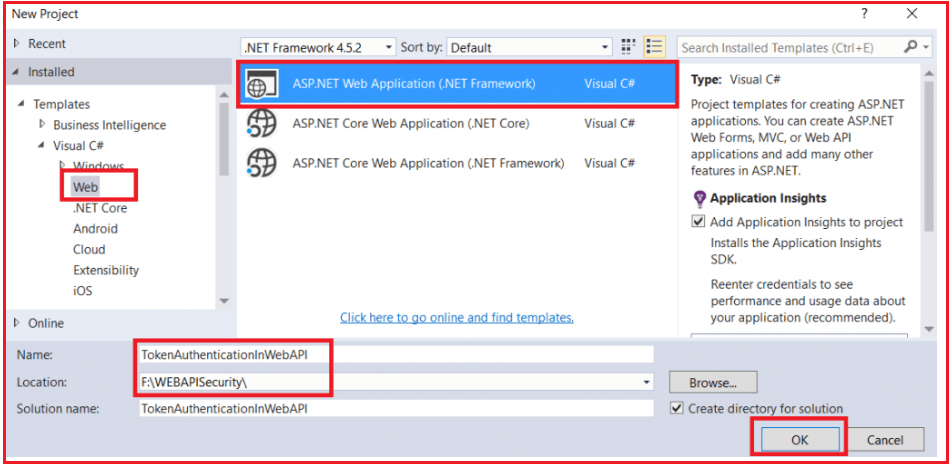
**INSERT** **INTO** UserMaster **VALUES**(103, 'Sambit', '123pqr', 'SuperAdmin', 'Sambit@g.com')

**INSERT** **INTO** UserMaster **VALUES**(104, 'Pranaya', 'abc123', 'Admin, User', 'Pranaya@g.com')

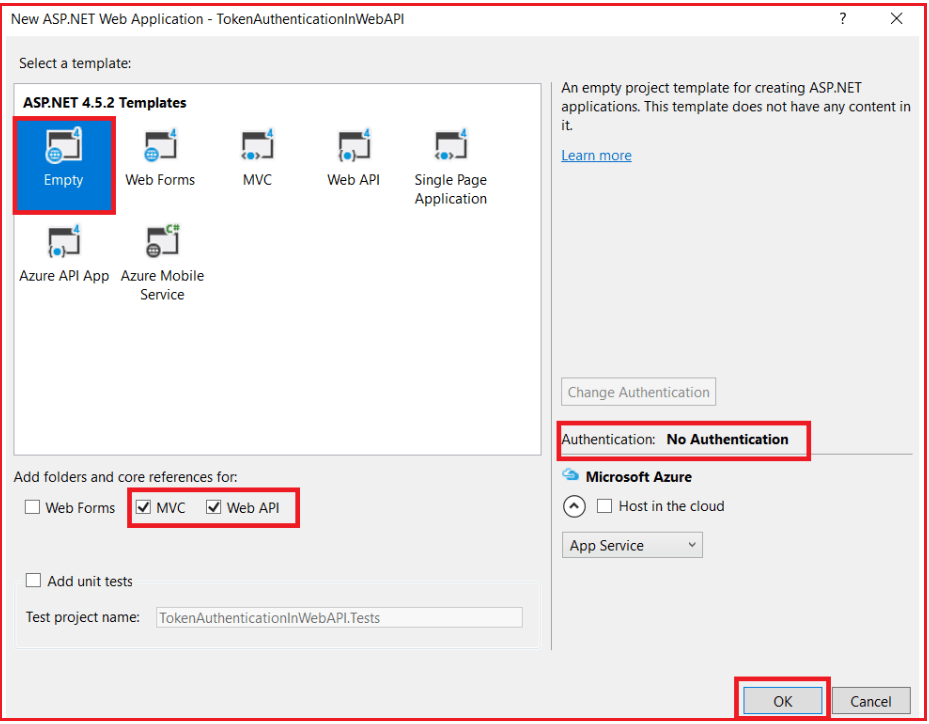
**GO**

**Step2: Creating an empty Web API Project with the name TokenAuthenticationWEBAPI**

Go to the **File menu > create > project** > here select “**asp.net web application**” under web. Provide the application name as **TokenAuthenticationWEBAPI** and select the project location where you want to create the project. Then click on the **OK** button as shown in the below image.



Once you click on the **OK** button, then a new window will open with Name **New ASP.NET Web Application** for selecting the **Project Templates** and from this window, you need to select the **Empty**project template as we are going to do everything from scratch and then checked the **MVC** and **Web API** checkbox from**Add folder and core references for** and then click on the **OK** button as shown in the below image.



**Step3: Add the required references from NuGet packages into your application.**

In order to Implement the **Token-Based Authentication in ASP.NET Web API**, we need to install the followings references from **NuGet packages.**Later part of this article, we will discuss the use of each the below packages.

1. **Microsoft.Owin.Host.SystemWeb**
2. **Microsoft.Owin.Security.OAuth**
3. **Microsoft.Owin.Cors**
4. **Newtonsoft.json**

For adding the above references from NuGet, Go to **Solution Explorer** > **Right Click on the References** > **Click on Manage NuGet Packages** > Search for the **Microsoft.Owin.Host.SystemWeb**,  **Microsoft.Owin.Security.OAuth,** **Microsoft.Owin.Cors and Newtonsoft.json**and install.

**Note:** When you install the above packages the dependency references are also automatically installed into your application.

**Step4: Creating the ADO.NET Entity Data Model**

Here we are going to use the **DB First Approach of Entity Framework** to create the Entity Data Model against the **SECURITY\_DB** database which we have already created and then select the **UserMaster** table from the **SECURITY\_DB**database.

**Step5: Create a Repository class**

Now, you need to create a class with the name **UserMasterRepository** which will validate the user and also returns the user information. As you can see in the below code, the **ValidateUser** method takes the username and password as input parameter and then validate this. If the username and password valid then it will return UserMaster object else it will return null. Later we will discuss when and where we will use this method.

**namespace** *TokenAuthenticationInWebAPI.Models*

**{**

**public** **class** UserMasterRepository : IDisposable

**{**

// SECURITY\_DBEntities it is your context class

SECURITY\_DBEntities context = new SECURITY\_DBEntities**()**;

//This method is used to check and validate the user credentials

**public** UserMaster ValidateUser**(**string username, string password**)**

**{**

**return** context.UserMasters.FirstOrDefault**(**user =**>**

user.UserName.Equals**(**username, StringComparison.OrdinalIgnoreCase**)**

&& user.UserPassword == password**)**;

**}**

**public** **void** Dispose**()**

**{**

context.Dispose**()**;

**}**

**}**

**}**

**Step6: Add a class for validating the user credentials asking for tokens.**

Now we need to add a class with the name **MyAuthorizationServerProvider** into our application. Within that class, we need to write the logic for **validating the user credentials** and **generating the access token**.

We need to inherit the **MyAuthorizationServerProvider** class from **OAuthAuthorizationServerProvider** class and then need to override the **ValidateClientAuthentication** and **GrantResourceOwnerCredentials** method. So, before proceeding and overriding these two methods, let us first understand what exactly these methods are going to perform.

**ValidateClientAuthentication Method:**

The **ValidateClientAuthentication** method is used for validating the client application. For the sake of simplicity, we will discuss what is a client and how to validate a client in more details in the next article.

**GrantResourceOwnerCredentials Method:**

The **GrantResourceOwnerCredentials** method is used to validate the client credentials (i.e. username and password). If it found the credentials are valid, then only it generates the access token. The client then using this access token can access the authorized resources from the Resource Server.

As we already discussed, the**signed access token** contains enough information to identify a user. Now the question is how. Let discuss this in details.

First, we need to create an instance of the **ClaimsIdentity** class and to the constructor of **ClaimsIdentity** class, we need to pass the **authentication type**. As we are going to use the **Token-Based Authentication**, so the Authentication Type is “**bearer token**”.

Once we create the **ClaimsIdentity** instance, then need to add the claims such as**Role, Name**, and **Email**, etc to the ClaimsIdentity instance. These are the user information which is going to be included in the **signed access token**. You can add any number of claims and once you add more claims. the token size will increase.

**MyAuthorizationServerProvider**

Create a class file with the name **MyAuthorizationServerProvider.cs** and then copy and paste the following code in it.

**using** *Microsoft.Owin.Security.OAuth;*

**using** *System.Security.Claims;*

**using** *System.Threading.Tasks;*

**namespace** *TokenAuthenticationInWebAPI.Models*

**{**

**public** **class** MyAuthorizationServerProvider : OAuthAuthorizationServerProvider

**{**

**public** **override** **async** Task ValidateClientAuthentication**(**OAuthValidateClientAuthenticationContext context**)**

**{**

context.Validated**()**;

**}**

**public** **override** **async** Task GrantResourceOwnerCredentials**(**OAuthGrantResourceOwnerCredentialsContext context**)**

**{**

**using** **(**UserMasterRepository \_repo = new UserMasterRepository**())**

**{**

var user = \_repo.ValidateUser**(**context.UserName, context.Password**)**;

**if** **(**user == **null)**

**{**

context.SetError**(**"invalid\_grant", "Provided username and password is incorrect"**)**;

**return**;

**}**

var identity = new ClaimsIdentity**(**context.Options.AuthenticationType**)**;

identity.AddClaim**(**new Claim**(**ClaimTypes.Role, user.UserRoles**))**;

identity.AddClaim**(**new Claim**(**ClaimTypes.Name, user.UserName**))**;

identity.AddClaim**(**new Claim**(**"Email", user.UserEmailID**))**;

context.Validated**(**identity**)**;

**}**

**}**

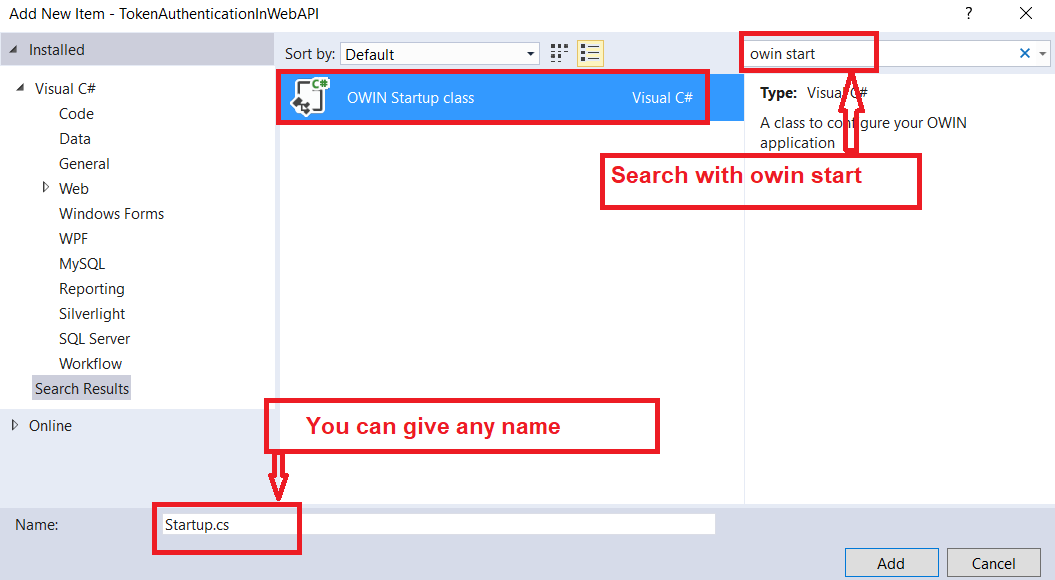
**}**

**}**

**Step7: Add the OWINStartup class.**

Now we need to add the **OWINStartup** class where we will configure the **OAuth Authorization Server**. This is going to be our authorization server.

To do so, go to the **Solution Explorer** > **Right Click on Project Name** form the Solution Explorer > **Add > New Item** > Select **OWIN Startup class** > Enter the class name as **Startup.cs** > and then click on the **Add** button as shown in the below image.



**Once you created the Owin Startup class, copy and paste the below code in it.**

**using** *System;*

**using** *Microsoft.Owin;*

**using** *Owin;*

**using** *TokenAuthenticationInWebAPI.Models;*

**using** *Microsoft.Owin.Security.OAuth;*

**using** *System.Web.Http;*

**[**assembly: OwinStartup**(**typeof**(**TokenAuthenticationInWebAPI.App\_Start.Startup**))]**

**namespace** *TokenAuthenticationInWebAPI.App\_Start*

**{**

// In this class we will Configure the OAuth Authorization Server.

**public** **class** Startup

**{**

**public** **void** Configuration**(**IAppBuilder app**)**

**{**

// Enable CORS (cross origin resource sharing) for making request using browser from different domains

app.UseCors**(**Microsoft.Owin.Cors.CorsOptions.AllowAll**)**;

OAuthAuthorizationServerOptions options = new OAuthAuthorizationServerOptions

**{**

AllowInsecureHttp = **true**,

//The Path For generating the Toekn

TokenEndpointPath = new PathString**(**"/token"**)**,

//Setting the Token Expired Time (24 hours)

AccessTokenExpireTimeSpan = TimeSpan.FromDays**(**1**)**,

//MyAuthorizationServerProvider class will validate the user credentials

Provider = new MyAuthorizationServerProvider**()**

**}**;

//Token Generations

app.UseOAuthAuthorizationServer**(**options**)**;

app.UseOAuthBearerAuthentication**(**new OAuthBearerAuthenticationOptions**())**;

HttpConfiguration config = new HttpConfiguration**()**;

WebApiConfig.Register**(**config**)**;

**}**

**}**

**}**

**Understanding the Owin Startup class code:**

Here we created a new instance of the **OAuthAuthorizationServerOptions** class and then set its options as follows:

1. Here, we set the path for generating the tokens as “**http://localhost:portnumber/token**”. Later we will see how to issue an HTTP Post request to generate the access token.
2. We have specified the expiry time for the access token as **24 hours**. So if the user tried to use the same access token after 24 hours from the issue time, then this request will be rejected and **HTTP status code 40**1 will be returned.
3. We also specified the implementation on how to **validate the client credentials for users asking for the access tokens** in the custom class named **MyAuthorizationServerProvider**.

Finally, we passed the **options** to the extension method **UseOAuthAuthorizationServer** which will add the authentication middleware to the pipeline.

**Step8: Add a Web API Controller.**

Now we need to create Web API resources. To do so, add an empty Web API Controller, where we will add some action methods so that we can check the **Token-Based Authentication** is working fine or not.

Go to **Solution Explorer** > **Right click on the Controllers** folder > **Add > Controller > Select WEB API 2 Controller – Empty** > Click on the **Add** button. > Enter the controller name as **TestController.cs** > finally click on the **Add** button which will create the TestController.

Once you created the **TestController**, then copy and paste the following code.

**using** *System.Linq;*

**using** *System.Security.Claims;*

**using** *System.Web.Http;*

**namespace** *TokenAuthenticationInWebAPI.Controllers*

**{**

**public** **class** TestController : ApiController

**{**

//This resource is For all types of role

**[**Authorize**(**Roles = "SuperAdmin, Admin, User"**)]**

**[**HttpGet**]**

**[**Route**(**"api/test/resource1"**)]**

**public** IHttpActionResult GetResource1**()**

**{**

var identity = **(**ClaimsIdentity**)**User.Identity;

**return** Ok**(**"Hello: " + identity.Name**)**;

**}**

//This resource is only For Admin and SuperAdmin role

**[**Authorize**(**Roles = "SuperAdmin, Admin"**)]**

**[**HttpGet**]**

**[**Route**(**"api/test/resource2"**)]**

**public** IHttpActionResult GetResource2**()**

**{**

var identity = **(**ClaimsIdentity**)**User.Identity;

var Email = identity.Claims

.FirstOrDefault**(**c =**>** c.Type == "Email"**)**.Value;

var UserName = identity.Name;

**return** Ok**(**"Hello " + UserName + ", Your Email ID is :" + Email**)**;

**}**

//This resource is only For SuperAdmin role

**[**Authorize**(**Roles = "SuperAdmin"**)]**

**[**HttpGet**]**

**[**Route**(**"api/test/resource3"**)]**

**public** IHttpActionResult GetResource3**()**

**{**

var identity = **(**ClaimsIdentity**)**User.Identity;

var roles = identity.Claims

.Where**(**c =**>** c.Type == ClaimTypes.Role**)**

.Select**(**c =**>** c.Value**)**;

**return** Ok**(**"Hello " + identity.Name + "Your Role(s) are: " + string.Join**(**",", roles.ToList**()))**;

**}**

**}**

**}**

Here, in the above controller, we have created three resources as follows,

1. **/api/test/resource1** – This resource can be accessed by all three types of roles such as Admin, SuperAdmin, and User
2. **/api/test/resource2** – This resource can be accessed by the users who are having the roles Admin and SuperAdmin
3. **/api/test/resource3** – This resource can be accessed only by the users who are having the role SuperAdmin

To test this we are going to use a client tool called **Postman**. First, you need to run your Web API application. If you are new to Postman then please read the following where we discussed how to use Postman to test Web API rest services.

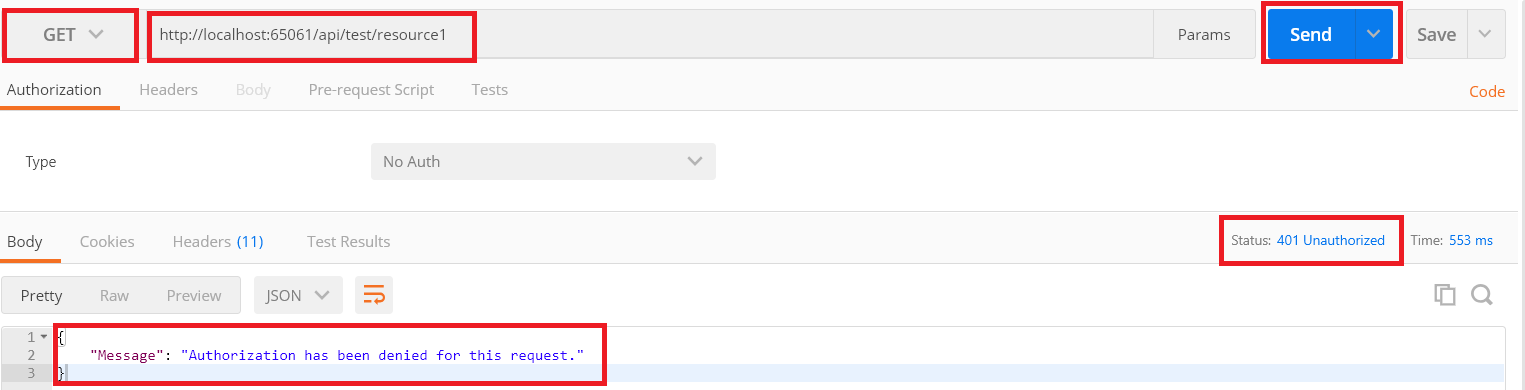
**[How to use Postman to test Rest Services?](https://dotnettutorials.net/lesson/how-to-use-postman-to-test-web-api/)**

**Step9: Testing the Token Authentication**

**Test1: Without Access Token, try to make a request for following URI**

**http://localhost:xxxxx/api/test/resource1**

You need to change the port number. You have to provide the port number where your Web API application is running.



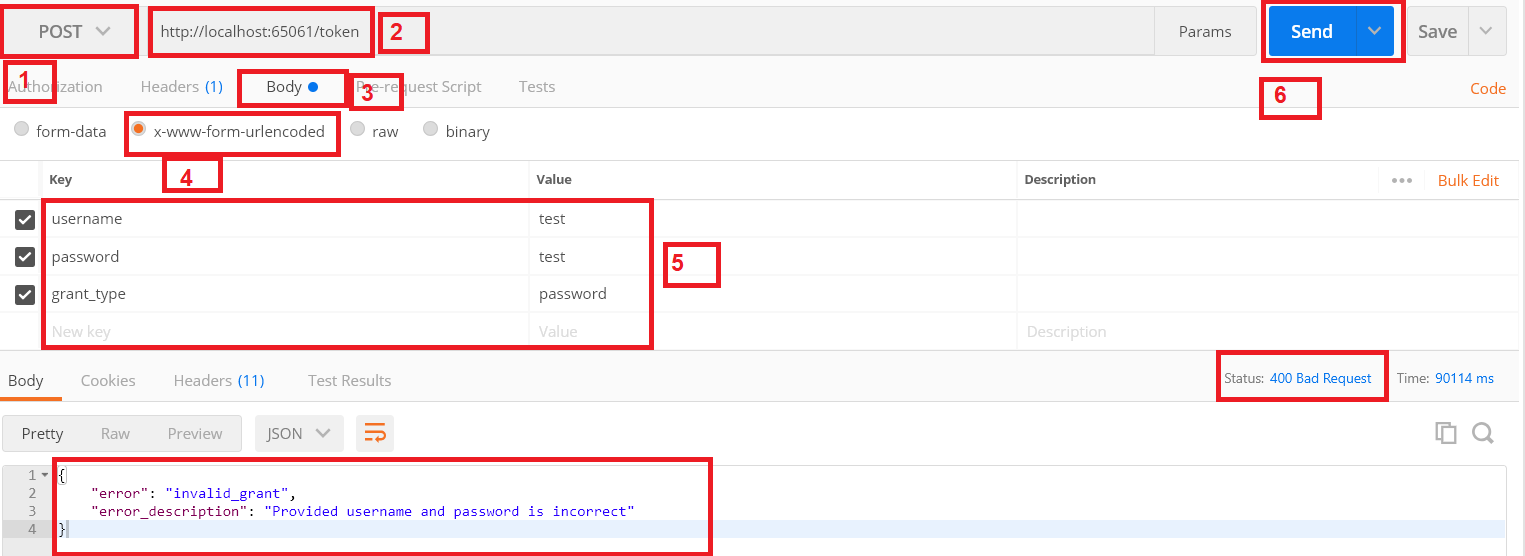
As expected you got 401 unauthorized responses

**Test2: Try to create the Access token with invalid credentials**

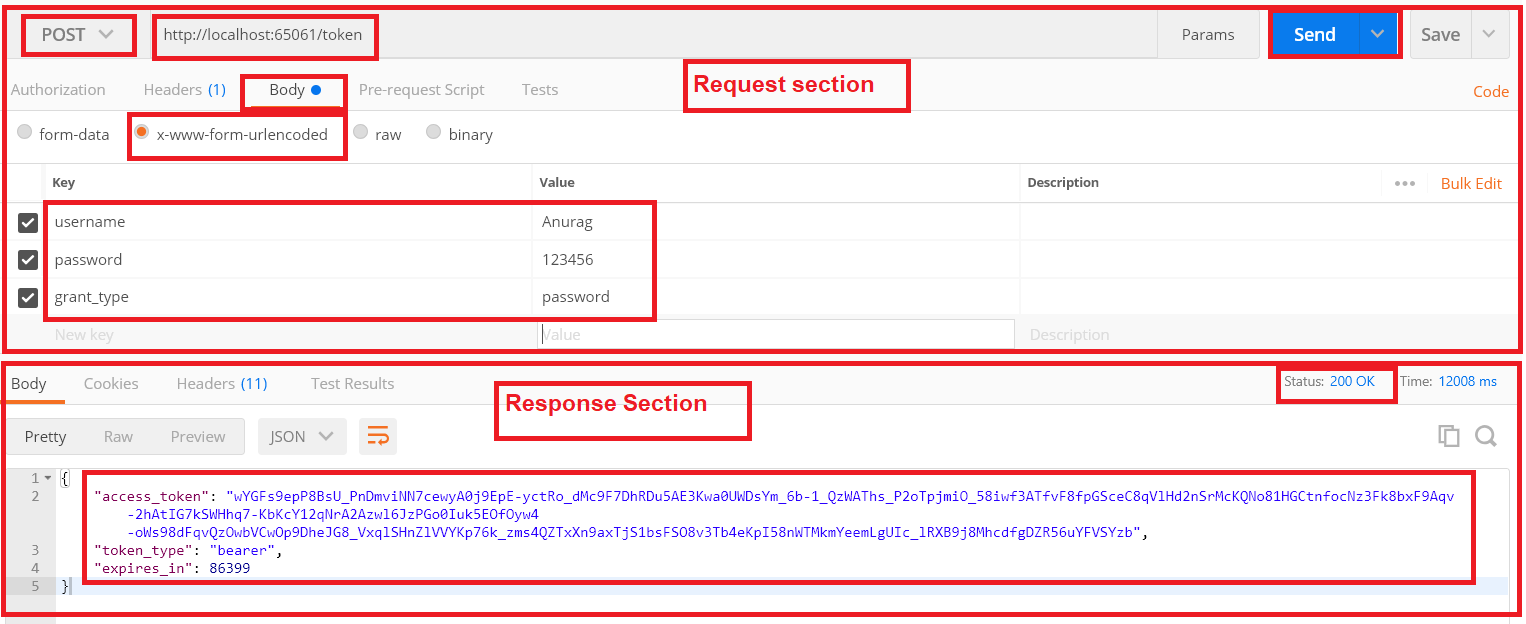
As we don’t have any user with the name test, so let’s try to create the Access Token for the test user. Select the method type as POST (1), enter the URL as **http://localhost:PortNumber/token** (2) and then click on body tab (3) and then select **x-www-form-urlencoded** (4) and then enter 3 parameters (5)

1. **username (value : test)**
2. **password (value: test)**
3. **grant\_type (value: password)**

And then click on the Send button (6).



Once you click on the send button, you will get status as 400 Bad Request as expected and it also tells that in the error description that the provided username and password are incorrect. Let’s generate the access token with valid credentials for the user Anurag whose password us 123456 as shown in the below image.



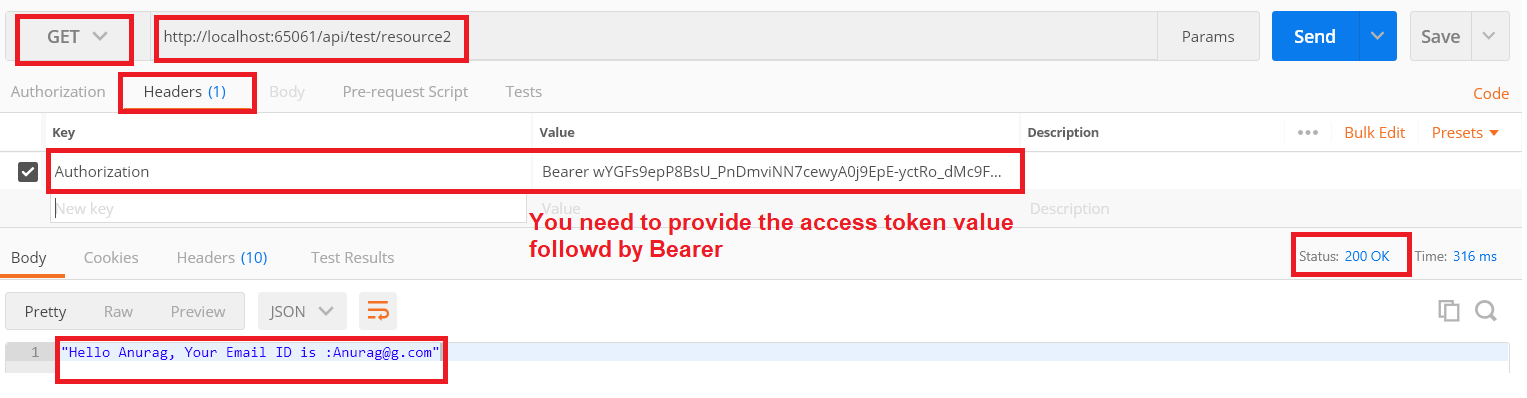
As you can see when you click on the send button, you will get status code 200 Ok along with you will get the access token which contains enough information to identify the user Anurag. You can also see in the Response section that the token type is Bearer and the token expire time in seconds.

**Test3: Access restricted resources with the access token.**

**/api/test/resource2**

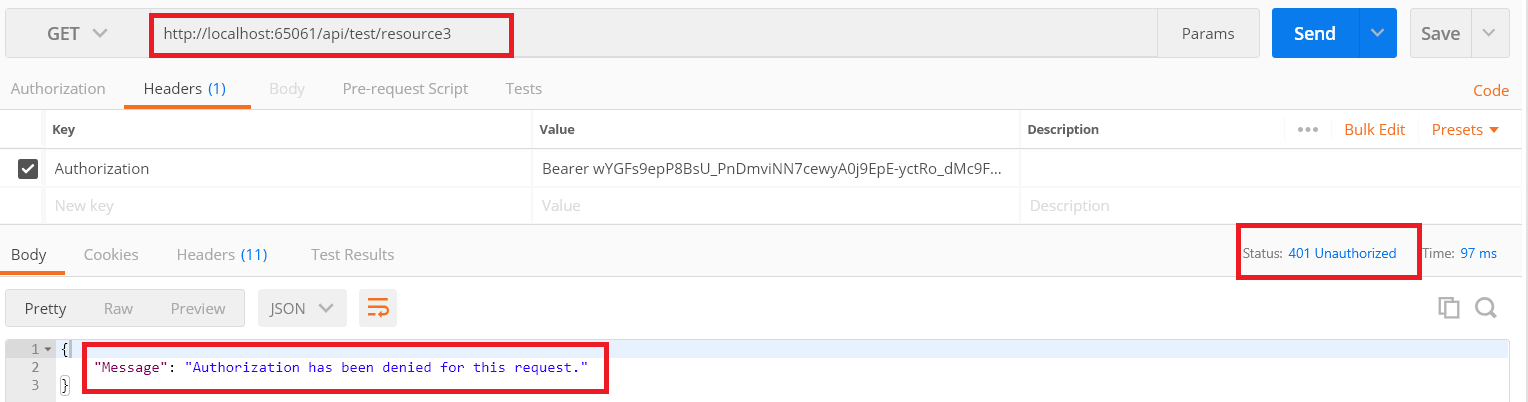
First copy the access token that we just generated in the previous example that we are going to use the token as shown below.

**Authorization: Bearer Access\_Token(value)**



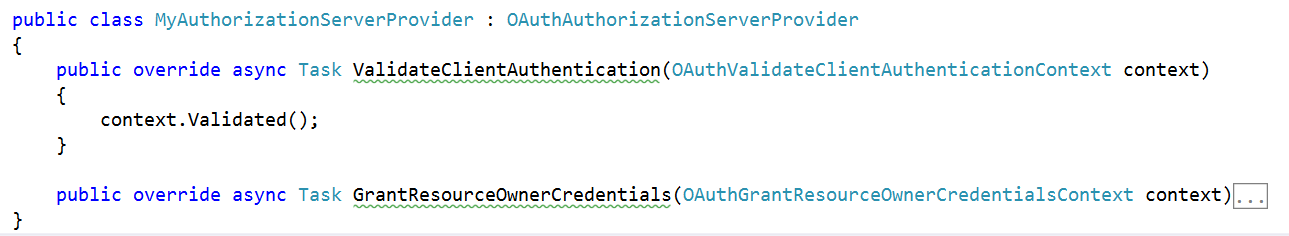
You can see that, when you click on the Send button, you will get 200 Ok as expected because the resource **/api/test/resource2** has been accessed by the Roles Admin and SuperAdmin and here the user Anurag has the Role Admin so, we get the above response.

But the above user cannot access the resource **/api/test/resource3** because the resource3 can only be accessed by the user whose role is SuperAdmin. Let’s prove this.



As you can see, the response is 401 unauthorized. But you generate the token for the user whose Role is SuperAdmin, then you can access this resource.

**Let’s have a look at the MyAuthorizationServiceProvider class**



The first method i.e. ValidateClientAuthentication method is responsible for validating the Client, in this example, we assume that we have only one client so we’ll always return that it is validated successfully. But in real-time you may have multiple clients and you need to validate the clients. So In the next article, we will discuss **[how to use the ValidateClientAuthentication method to validate the client](https://dotnettutorials.net/lesson/client-validation-using-basic-authentication-web-api/)**.

**Advantages of using Token Based Authentication in ASP.NET Web API:**

**Scalability of Servers:**

The token which is sent to the server by the client is self-contained means it holds enough data to identify the user needed for authentication. As a result, you can add easily more servers to your web farm, there is no dependent on shared session stores.

**Loosely Coupling:**

The client application is not tied or coupled with any specific authentication mechanism. The token is generated, validated and perform the authentication are done by the server only.

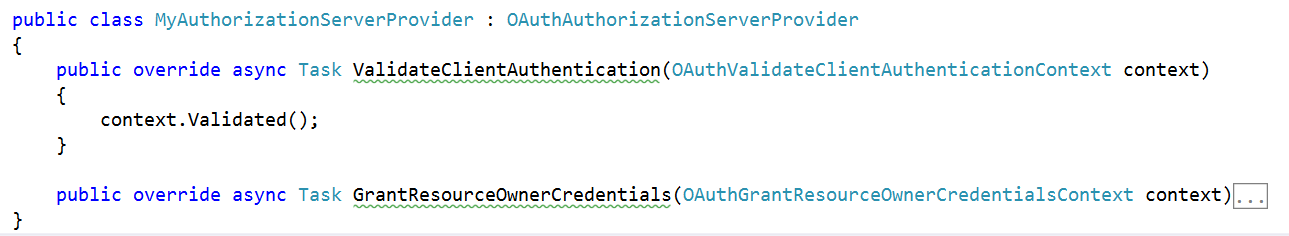
**Mobile-Friendly:**

The Cookies and browsers like each other, but handling the cookies on native platforms like Android, iOS, Windows Phone is not an easy task. The token-based approach simplifies this a lot.

**Client Validation Using Basic Authentication in Web API**

In this article, I am going to discuss how to implement **Client Validation Using Basic Authentication in Web API**. Please read our previous article before proceeding to this article as we are going to work the same example. In our last article, we discussed how to implement **[Token Based Authentication in ASP.NET Web API](https://dotnettutorials.net/lesson/token-based-authentication-web-api/)**.

If you observed in the last article, we have created the following MyAuthorizationServiceProvider class.

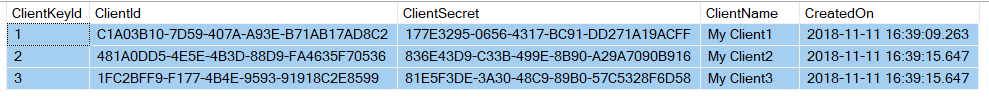


The first method i.e. ValidateClientAuthentication method is responsible for validating the Client, in the above example, we assume that we have only one client so we’ll always return that it is validated successfully.

Let’s change the requirement. Assume that we have more than one client, who is going to consume our service. In such a case, we need to validate the clients within the ValidateClientAuthentication method.

**Let’s see how to achieve this.**

For this, we are going to use the following ClientMaster table



**Please use below SQL Script to create and populate the ClientMaster table with some test data.**

**USE** SECURITY\_DB

**GO**

-- Create ClientMaster table

**CREATE** **TABLE** ClientMaster

(

ClientKeyId **INT** **PRIMARY** **KEY** **IDENTITY**,

ClientId **VARCHAR**(500),

ClientSecret **VARCHAR**(500),

ClientName **VARCHAR**(100),

CreatedOn DateTime

)

**GO**

-- Populate the ClientMaster with test data

**INSERT** **INTO** ClientMaster(ClientId, ClientSecret, ClientName, CreatedOn)

**VALUES**(**NEWID**(), **NEWID**(), 'My Client1', **GETDATE**())

**INSERT** **INTO** ClientMaster(ClientId, ClientSecret, ClientName, CreatedOn)

**VALUES**(**NEWID**(), **NEWID**(), 'My Client2', **GETDATE**())

**INSERT** **INTO** ClientMaster(ClientId, ClientSecret, ClientName, CreatedOn)

**VALUES**(**NEWID**(), **NEWID**(), 'My Client3', **GETDATE**())

Once you create the ClientMaster table, then you need to update the EDMX file to add the above ClientMaster table.

**Create a class file with the name ClientMasterRepository.cs and then copy and paste the following code.**

**namespace** *TokenAuthenticationInWebAPI.Models*

**{**

**public** **class** ClientMasterRepository : IDisposable

**{**

// SECURITY\_DBEntities it is your context class

SECURITY\_DBEntities context = new SECURITY\_DBEntities**()**;

//This method is used to check and validate the Client credentials

**public** ClientMaster ValidateClient**(**string ClientID, string ClientSecret**)**

**{**

**return** context.ClientMasters.FirstOrDefault**(**user =**>**

user.ClientId == ClientID

&& user.ClientSecret == ClientSecret**)**;

**}**

**public** **void** Dispose**()**

**{**

context.Dispose**()**;

**}**

**}**

**}**

Here we create the ValidateClient method which is very straightforward. It’s the ClientID and ClientSecret as input parameter and checks in the ClientMaster table whether the client is valid or not and it simply returns the client details.

**Now we need to modify the ValidateClientAuthentication() method of MyAuthorizationServerProvider class as shown below.**

**public** **override** **async** Task ValidateClientAuthentication**(**OAuthValidateClientAuthenticationContext context**)**

**{**

string clientId = string.Empty;

string clientSecret = string.Empty;

// The TryGetBasicCredentials method checks the Authorization header and

// Return the ClientId and clientSecret

**if** **(**!context.TryGetBasicCredentials**(**out clientId, out clientSecret**))**

**{**

context.SetError**(**"invalid\_client", "Client credentials could not be retrieved through the Authorization header."**)**;

context.Rejected**()**;

**return**;

**}**

//Check the existence of by calling the ValidateClient method

ClientMaster client = **(**new ClientMasterRepository**())**.ValidateClient**(**clientId, clientSecret**)**;

**if** **(**client != **null)**

**{**

// Client has been verified.

context.OwinContext.Set**<**ClientMaster**>(**"oauth:client", client**)**;

context.Validated**(**clientId**)**;

**}**

**else**

**{**

// Client could not be validated.

context.SetError**(**"invalid\_client", "Client credentials are invalid."**)**;

context.Rejected**()**;

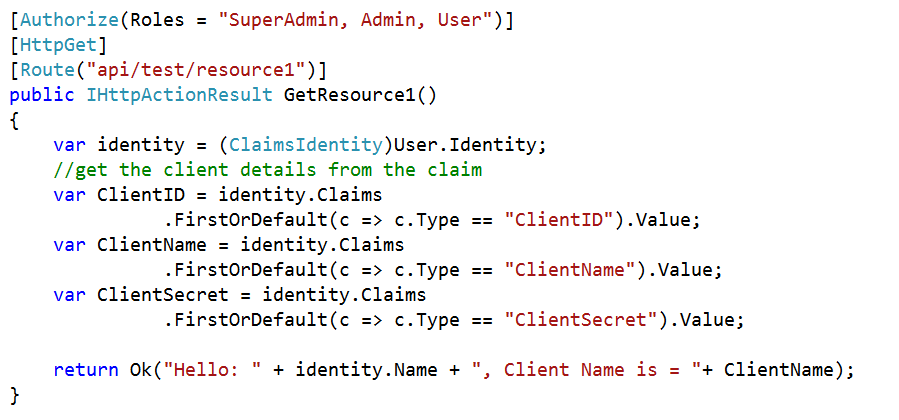
**}**

context.Validated**()**;

**}**

**Note:** We need to pass the ClientId and ClientSecret using the Basic authentication in the authorization header i.e. in Base64 encoded format.

**Modify the GetResource1 action method of the TestController as shown below.**

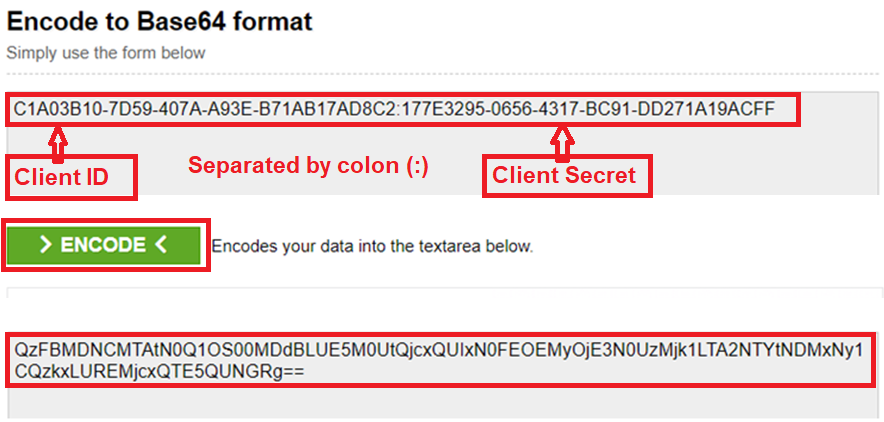


**Testing the API using Postman:**

Let’s first create the Base64 Encode value by for the ClientID and ClientSecret by using the following website

<https://www.base64encode.org/>

Enter the ClientID and ClientSecret separated by a colon (:) in **“Encode to Base64 format”** textbox, and then click on the **“Encode”**button as shown in the below diagram which will generate the Base64 encoded value.



Once you generate the Base64 encoded string, let’s see how to use basic authentication in the header to pass the Base64 encoded value.

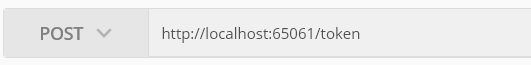
Here we need to use the Authorization header and the value will be the Base64 encoded string followed the “BASIC” as shown below.

**Authorization: BASIC QzFBMDNCMTAtN0Q1OS00MDdBLUE5M0UtQjcxQUIxN0FEOEMyOjE3N0UzMjk1LTA2NTYtNDMxNy1CQzkxLUREMjcxQTE5QUNGRg==**

**Let’s see step by step procedure to use the Postman to generate the Access Token**

**Step1:**

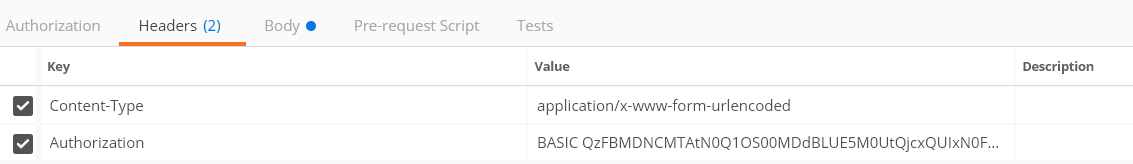
Select the Method as POST and provide URI as shown below in the below image



**Step2:**

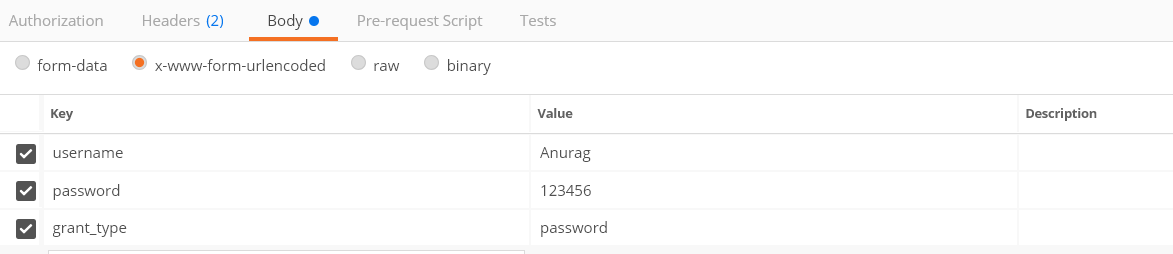
Select the Header tab and provide the Authorization value as shown below.

Authorization: BASIC QzFBMDNCMTAtN0Q1OS00MDdBLUE5M0UtQjcxQUIxN0FEOEMyOjE3N0UzMjk1LTA2NTYtNDMxNy1CQzkxLUREMjcxQTE5QUNGRg==

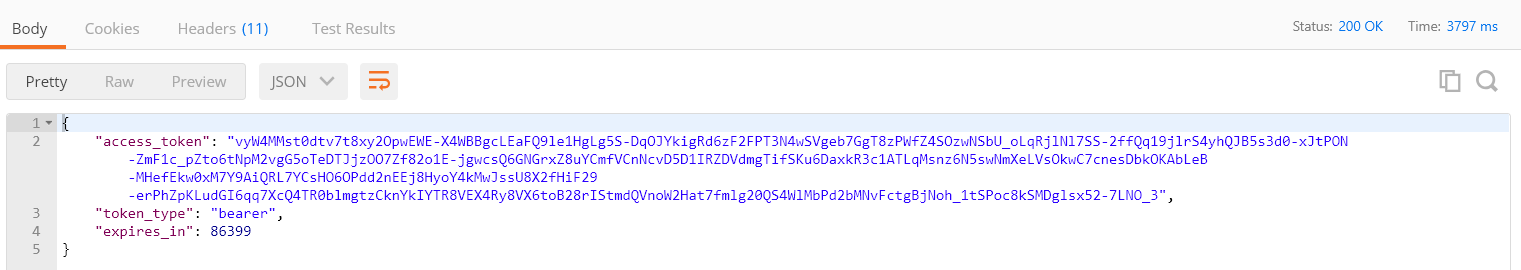


**Step3:**

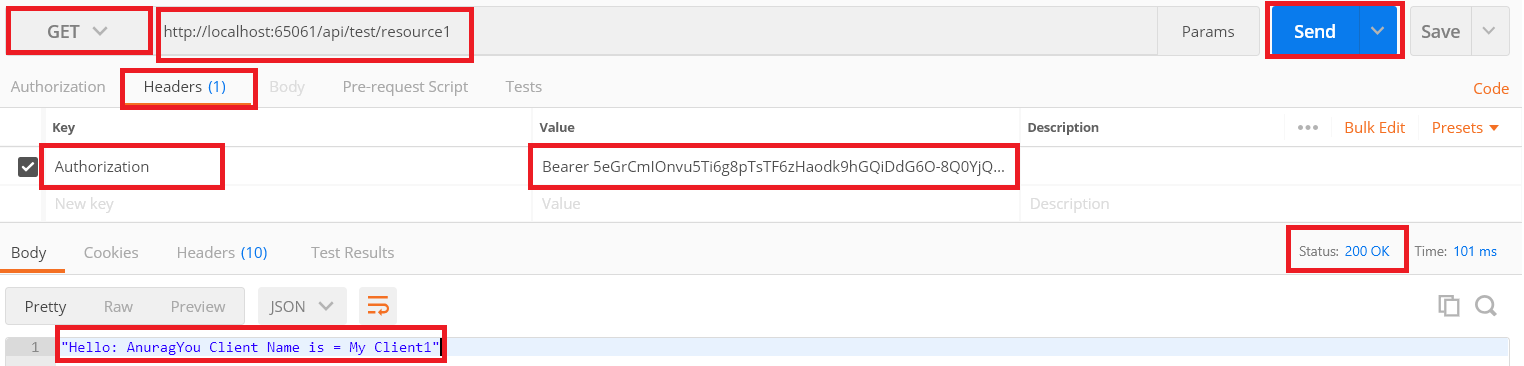
Select the Body Tab. Then choose x-www-form-urlencoded option and provide the username and password value. Provide the grant\_type value as password as shown in the below image,



Now click on the Send button which will generate the access token as shown below.



Once the access token is generated, we use that token to access the resources as shown below.



**Refresh Token in Web API**

In this article, I am going to discuss how to implement **Refresh Token in Web API** by validating the clients as well as I will also discuss how to persist the refresh token into a database. Please read the following two articles before proceeding to this article as we are going to use the same example that we worked with in our previous two articles.

**[Token Based Authentication in Web API](https://dotnettutorials.net/lesson/token-based-authentication-in-web-api/)**: In this article, we discussed how to implement and use the Token Based Authentication in Web API.

**[Client Validation in Token Based Authentication](https://dotnettutorials.net/lesson/client-validation-using-basic-authentication-web-api/)**: In this article, we discussed how to validate the clients while generating the token in Web API.

**What is a Refresh Token?**

A **Refresh Token** is a special kind of token that can be used to obtain a new renewed access token which allows access to the protected resources. You can request for the new access tokens by using the Refresh Token in Web API until the Refresh Token is blacklisted.

**Why we need Refresh Token in Web API?**

The idea of using the refresh token is to issue a short-lived access token (up to 30 minutes) for the first time and then use the refresh token to obtain a new access token and use that access token to access the protected resources.

So, the user needs to provide the username and password along with the client info (i.e. the client id and client secret) to authenticate himself, and if the information provided by the user is valid, then a response contains a short-lived access token along with a long-lived refresh token gets generated.

The refresh token is not an access token it is just an identifier for the access token**.** Now once the access token is expired, the user can use the refresh token to obtain another short-lived access token and so on.

**Why not long-lived access token?**

Now, you may have one question in your mind. Why not we are issuing a long-lived access token for the first time?

Let’s discuss why not a long-lived access token or what are the advantages of using refresh token in Web API. Mainly there are three main reasons to use the refresh tokens are as follows

**Updating the Access Token Content**:

As we already discussed, the access tokens are self-contained tokens means they contain all the information (which is known as claims) of an authenticated user once the access token is generated.

Now, if we issue a long-lived access token, let say for example 1 month, for a user let’s say “Anurag” and let say the user “Anurag” is enrolled with the role “Users” at the moment. So all these information get stored on the access token which is generated by the Authorization server.

If you have decided (3 days after he obtained the access token) to add him with the role “Admin” and then there is no way to update this information in the access token which is already generated, you need to ask him to re-authenticate himself again, so that the Authorization server add the updated information to the newly generated access token, and this is not feasible in most of the cases. You might not be able to reach to the users who already obtained the long-lived access tokens.

So to overcome the above issue, you need to issue short-lived access token (30 minutes for example) along with a long-lived refresh token and then the user needs to use the refresh token to obtain the newly updated access token, once the user obtains the new access token, the Authorization Server will be able to add the updated claims or new claims to the new access token being generated.

**Revoking the Access from Authenticated users**:

Once the user obtained the long-lived access token, then he will be able to access the server resources as long as his access token is not expired and there is no standard way to revoke the access tokens unless and until the Authorization Server implements some custom logic to store the generated access token a database and need to do database checks with each and every request.

But with the refresh token, a database or system admin can simply revoke the access by deleting the refresh token identifier from the database. So, when the user requests for a new access token by using the deleted refresh token, the Authorization Server will reject this request because the refresh token is no longer available in the database.

**No need to store or ask for the username and password frequently**:

Using refresh token allows you to ask the user for his username and password only one time (i.e. for the first time), then the Authorization Server can issue very long-lived refresh token (1 year for example) and the user will stay logged in all this period until and unless system admin tries to revoke (delete) the refresh token. This can be very useful if you are building an API which will be consumed by a front-end application where it is not feasible to keep asking for the username/password frequently.

So for the above three major reasons we need to use Refresh Tokens.

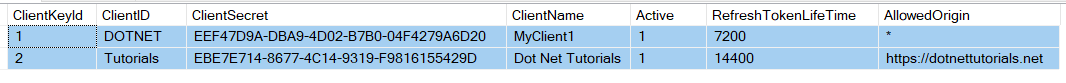
**The Refresh Tokens and Clients**

In order to use the refresh token, we need to be bound the refresh token with a **Client.** In simple word, we can define a client as an application who wants to access our resources. Each Client should have a unique Client Id and Client Secret.

The Client Id is an unique public information which identifies the application among other applications. The client id can be included in the source code of your application, but the client secret must stay confidential.

Bounding the refresh token to a client is very important this is because you do not want any refresh token generated by your Authorization Server to be used by another client to obtain the access token.

The schema for the client’s table should be as shown below.



In our previous article, we create the **ClientMaster** table, so let’s delete the existing **ClientMaster** table and regenerate the **ClientMaster** table with the above structure.

Please use below SQL Script to DROP, Create and Populate the ClientMaster table with two different clients which we are going to use in this demo.

**USE** [SECURITY\_DB]

**GO**

--First DROP ClientMaster table

**DROP** **TABLE** ClientMaster

-- Create the ClientMaster table with new structure

**CREATE** **TABLE** [ClientMaster](

[ClientKeyId] **INT** **PRIMARY** **KEY** **IDENTITY**(1,1),

[ClientID] **VARCHAR**(500) NOT **NULL**,

[ClientSecret] **VARCHAR**(500) NOT **NULL**,

[ClientName] **VARCHAR**(100) NOT **NULL**,

[Active] **BIT** NOT **NULL**,

[RefreshTokenLifeTime] **INT** NOT **NULL**,

[AllowedOrigin] **VARCHAR**(500) NOT **NULL**

)

**GO**

-- INSERT the client details inti the ClientMaster table

**INSERT** **INTO** ClientMaster **VALUES**('DOTNET',**NEWID**(),'MyClient1',1,7200,'\*')

**INSERT** **INTO** ClientMaster **VALUES**('Tutorials',**NEWID**(),'Dot Net Tutorials',1,14400,'https://dotnettutorials.net')

**GO**

**Let’s us discuss the use of each column of ClientMaster table**

The **ClientID** and **ClientSecret** columns of the ClientMaster table uniquely identify a particular client.

The **Active** column is also very important; if the system admin is decided to deactivate a particular client so that any new requests asking for the access token from that particular deactivated client will be rejected by the Authorization Server.

The **Refresh Token Life Time** column is used to set when the refresh token (not the access token) will expire in minutes.

Finally, the **Allowed Origin** column is used to configure the CORS and to set “**Access-Control-Allow-Origin**” on the back-end API.

**Refresh Token Schema:**

As we already discussed, we need to store the refresh tokens generated by the Authorization Server into a database and this is very important to facilitate the management for refresh tokens. The schema for the **Refresh Token** table as shown in the below image:

Refresh Token in Web API

**Please use below SQL Script to create the RefershToken table.**

-- Create the RefreshToken table

**CREATE** **TABLE** RefreshToken

(

[**ID**] **VARCHAR**(500) **PRIMARY** **KEY**,

[UserName] **VARCHAR**(500),

[ClientID] **VARCHAR**(500),

[IssuedTime] **DATETIME**,

[ExpiredTime] **DATETIME**,

[ProtectedTicket] **VARCHAR**(500)

)

**Let discuss the use of each column of the RefreshToken table.**

The **ID** column of the RefreshToken table contains the hashed value of the refresh token id, the API consumer will receive and send the plain refresh token Id and the **UserName**column indicates to which user this refresh token belongs, and the same thing is applied for **ClientID**column indicating that the Token belongs to that particular clients.

By having the **ClientID** column, as a system admin, you can revoke (delete) the refresh token for certain user on the certain client and keep the other refresh tokens for the same user obtained by different clients. For example, let say you have two clients having the same username, if you delete one user for a particular client, then the same user of other clients can access the refresh token.

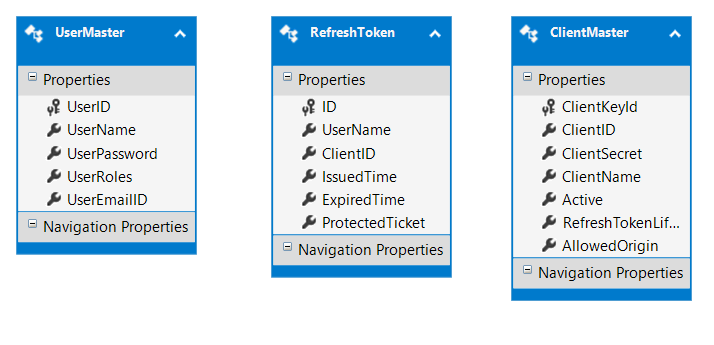
The **IssuedTime**and **ExpiredTime**columns are for displaying purpose only.

Finally, the **Protected Ticket** column contains the magical signed string which contains a serialized representation for the ticket for a specific user, in other words, it contains all the claims and ticket properties for a user.

We have discussed enough theory, so it’s the time to put all the theories into practice. So let’s discuss the step by step procedure to implement the Refresh Token in Web API. As I already told you that we are going to use the same example that we worked with our previous two articles.

**Step1: Modify the EDMX file**

We need to modify the EDMX file to add the newly generated RefreshToken table and we also need to update the ClientMaster table. Once you modify your EDMX file, the EDMX file should look as shown below.



**Step2: Modify the ClientMasterRepository class as shown below**

**namespace** *TokenAuthenticationInWebAPI.Models*

**{**

**public** **class** ClientMasterRepository : IDisposable

**{**

// SECURITY\_DBEntities it is your context class

SECURITY\_DBEntities context = new SECURITY\_DBEntities**()**;

//This method is used to check and validate the Client credentials

**public** ClientMaster ValidateClient**(**string ClientID, string ClientSecret**)**

**{**

**return** context.ClientMasters.FirstOrDefault**(**user =**>**

user.ClientID == ClientID

&& user.ClientSecret == ClientSecret**)**;

**}**

**public** **void** Dispose**()**

**{**

context.Dispose**()**;

**}**

**}**

**}**

**Step3: Adding a Helper class**

Add a class file with the name Helper.cs and then copy and paste the following code

**using** *System;*

**using** *System.Security.Cryptography;*

**namespace** *TokenAuthenticationInWebAPI.Models*

**{**

**public** **class** Helper

**{**

**public** **static** string GetHash**(**string input**)**

**{**

HashAlgorithm hashAlgorithm = new SHA256CryptoServiceProvider**()**;

**byte[]** byteValue = System.Text.Encoding.UTF8.GetBytes**(**input**)**;

**byte[]** byteHash = hashAlgorithm.ComputeHash**(**byteValue**)**;

**return** Convert.ToBase64String**(**byteHash**)**;

**}**

**}**

**}**

The above GetHash method is straightforward; it takes an input of string type and returns its hash value.

**Step4: Add AuthenticationRepository class file**

Add a class file with the name **AuthenticationRepository.cs** and then copy and paste the following code.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *System.Threading.Tasks;*

**namespace** *TokenAuthenticationInWebAPI.Models*

**{**

**public** **class** AuthenticationRepository : IDisposable

**{**

SECURITY\_DBEntities context = new SECURITY\_DBEntities**()**;

//Add the Refresh token

**public** **async** Task**<bool>** AddRefreshToken**(**RefreshToken token**)**

**{**

var existingToken = context.RefreshTokens.FirstOrDefault**(**r =**>** r.UserName == token.UserName

&& r.ClientID == token.ClientID**)**;

**if** **(**existingToken != **null)**

**{**

var result = **await** RemoveRefreshToken**(**existingToken**)**;

**}**

context.RefreshTokens.Add**(**token**)**;

**return** **await** context.SaveChangesAsync**()** **>** 0;

**}**

//Remove the Refesh Token by id

**public** **async** Task**<bool>** RemoveRefreshTokenByID**(**string refreshTokenId**)**

**{**

var refreshToken = **await** context.RefreshTokens.FindAsync**(**refreshTokenId**)**;

**if** **(**refreshToken != **null)**

**{**

context.RefreshTokens.Remove**(**refreshToken**)**;

**return** **await** context.SaveChangesAsync**()** **>** 0;

**}**

**return** **false**;

**}**

//Remove the Refresh Token

**public** **async** Task**<bool>** RemoveRefreshToken**(**RefreshToken refreshToken**)**

**{**

context.RefreshTokens.Remove**(**refreshToken**)**;

**return** **await** context.SaveChangesAsync**()** **>** 0;

**}**

//Find the Refresh Token by token ID

**public** **async** Task**<**RefreshToken**>** FindRefreshToken**(**string refreshTokenId**)**

**{**

var refreshToken = **await** context.RefreshTokens.FindAsync**(**refreshTokenId**)**;

**return** refreshToken;

**}**

//Get All Refresh Tokens

**public** List**<**RefreshToken**>** GetAllRefreshTokens**()**

**{**

**return** context.RefreshTokens.ToList**()**;

**}**

**public** **void** Dispose**()**

**{**

context.Dispose**()**;

**}**

**}**

**}**

The methods we add in the above **AuthenticationRepository** class will add support for manipulating the **RefreshToken** table that we have added, they are self-explanatory methods and there is nothing special about them.

**Step5: Modify the Client Validation logic**

Here, we need to modify the logic responsible for validating the client information whether the request needs an access token or uses a refresh token to obtain a new access token. So modify the **ValidateClientAuthentication** method of the **MyAuthorizationServerProvider** class as shown below.

**public** **override** Task ValidateClientAuthentication**(**OAuthValidateClientAuthenticationContext context**)**

**{**

string clientId = string.Empty;

string clientSecret = string.Empty;

// The TryGetBasicCredentials method checks the Authorization header and

// Return the ClientId and clientSecret

**if** **(**!context.TryGetBasicCredentials**(**out clientId, out clientSecret**))**

**{**

context.SetError**(**"invalid\_client", "Client credentials could not be retrieved through the Authorization header."**)**;

**return** Task.FromResult**<object>(null)**;

**}**

//Check the existence of by calling the ValidateClient method

ClientMaster client = **(**new ClientMasterRepository**())**.ValidateClient**(**clientId, clientSecret**)**;

**if** **(**client == **null)**

**{**

// Client could not be validated.

context.SetError**(**"invalid\_client", "Client credentials are invalid."**)**;

**return** Task.FromResult**<object>(null)**;

**}**

**else**

**{**

**if** **(**!client.Active**)**

**{**

context.SetError**(**"invalid\_client", "Client is inactive."**)**;

**return** Task.FromResult**<object>(null)**;

**}**

// Client has been verified.

context.OwinContext.Set**<**ClientMaster**>(**"ta:client", client**)**;

context.OwinContext.Set**<**string**>(**"ta:clientAllowedOrigin", client.AllowedOrigin**)**;

context.OwinContext.Set**<**string**>(**"ta:clientRefreshTokenLifeTime", client.RefreshTokenLifeTime.ToString**())**;

context.Validated**()**;

**return** Task.FromResult**<object>(null)**;

**}**

**}**

**Explanation of the above code**

We are trying to get the Client ID and Client Secret from the authorization header using a basic scheme, so the user needs to send the Client ID and Client Secret in the base64 encode format (client\_id:client\_secret) and need to send it in the Authorization header of the HTTP Request.

Once we receive the client id and client secret, we need to check it in our database whether the client is already registered with our back-end API or not (means we need to validate the client), if it is not registered we will invalidate the context and reject the request.

If the client is registered, then we will check whether the client is active or not, if it is not active, then we will also invalidate the context and reject the request

And, if we found the client is active, then we need to store the **client allowed origin** and **refresh token lifetime** value on the Owin context (if you want then you can store all information of the client), so it will be available once we generate the refresh token.

If all is valid we mark the context as a valid context which means that client validation has passed and the flow can proceed to the next step.

**Step6: Validating the Resource Owner Credentials**

Once the client validation has been passed, next we need to validate the resource owner credentials i.e. the username and password are correct or not and then we need to bound the client id to the accession generated. To do so, let’s modify the GrantResourceOwnerCredentials method of the MyAuthorizationServerProvider class as shown below.

**public** **override** **async** Task GrantResourceOwnerCredentials**(**OAuthGrantResourceOwnerCredentialsContext context**)**

**{**

ClientMaster client = context.OwinContext.Get**<**ClientMaster**>(**"ta:client"**)**;

var allowedOrigin = context.OwinContext.Get**<**string**>(**"ta:clientAllowedOrigin"**)**;

**if** **(**allowedOrigin == **null)**

**{**

allowedOrigin = "\*";

**}**

context.OwinContext.Response.Headers.Add**(**"Access-Control-Allow-Origin", new**[]** **{** allowedOrigin **})**;

UserMaster user = **null**;

**using** **(**UserMasterRepository \_repo = new UserMasterRepository**())**

**{**

user = \_repo.ValidateUser**(**context.UserName, context.Password**)**;

**if** **(**user == **null)**

**{**

context.SetError**(**"invalid\_grant", "Provided username and password is incorrect"**)**;

**return**;

**}**

**}**

var identity = new ClaimsIdentity**(**context.Options.AuthenticationType**)**;

identity.AddClaim**(**new Claim**(**ClaimTypes.Role, user.UserRoles**))**;

identity.AddClaim**(**new Claim**(**ClaimTypes.Name, user.UserName**))**;

identity.AddClaim**(**new Claim**(**"Email", user.UserEmailID**))**;

var props = new AuthenticationProperties**(**new Dictionary**<**string, string**>**

**{**

**{**

"client\_id", **(**context.ClientId == **null)** ? string.Empty : context.ClientId

**}**,

**{**

"userName", context.UserName

**}**

**})**;

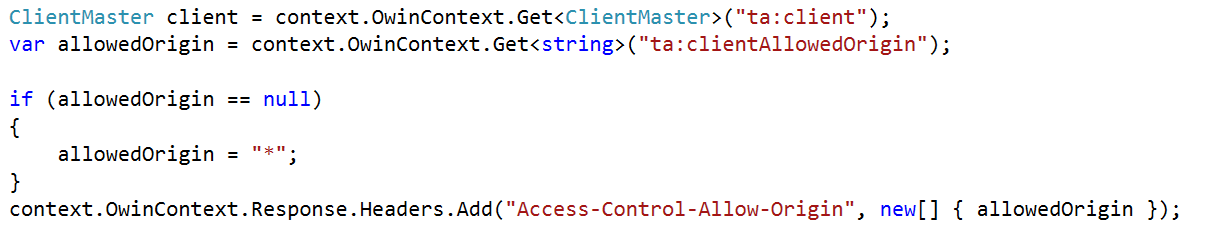
var ticket = new AuthenticationTicket**(**identity, props**)**;

context.Validated**(**ticket**)**;

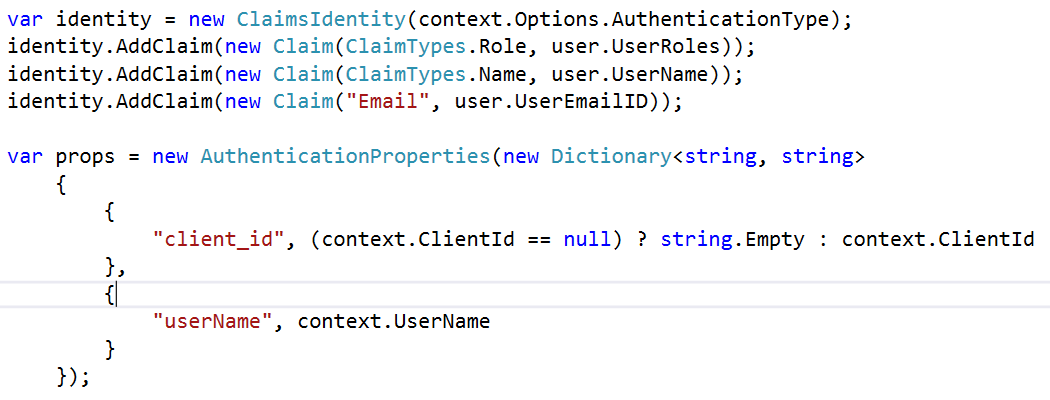
**}**

**Explanation of the Above Code:**

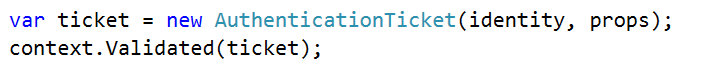
First, we need to read the client information from the Owin context and then we add the clientAllowedOrigin value to add the header “Access-Control-Allow-Origin” to Owin context response as shown in the below image.



Then we will check the username and password for the resource owner and if it is valid, then we will generate the set of claims for the above user along with authentication properties which contain the client id and username as shown in the below image.



Now the access token will be generated behind the scenes when we call the context.Validated(ticket) method as shown in the below image.



**Step8: Implementing the TokenEndpoint method**

Now we need to override the TokenEndpoint method within the MyAuthorizationServerProvider class with the following code.

**public** **override** Task TokenEndpoint**(**OAuthTokenEndpointContext context**)**

**{**

**foreach** **(**KeyValuePair**<**string, string**>** property in context.Properties.Dictionary**)**

**{**

context.AdditionalResponseParameters.Add**(**property.Key, property.Value**)**;

**}**

**return** Task.FromResult**<object>(null)**;

**}**

The above TokenEndpoint method is adding some additional properties to the token response.

**Step9: Generating Refresh Token in Web API and persisting it into a database**

Now we need to generate the Refresh Token and Store it into our database inside the RefreshToken table. To do so, add a class file with the name RefreshTokenProvider.cs under the Models folder and then copy and paste the following code.

**using** *Microsoft.Owin.Security.Infrastructure;*

**using** *System;*

**using** *System.Threading.Tasks;*

**namespace** *TokenAuthenticationInWebAPI.Models*

**{**

**public** **class** RefreshTokenProvider : IAuthenticationTokenProvider

**{**

**public** **async** Task CreateAsync**(**AuthenticationTokenCreateContext context**)**

**{**

//Get the client ID from the Ticket properties

var clientid = context.Ticket.Properties.Dictionary**[**"client\_id"**]**;

**if** **(**string.IsNullOrEmpty**(**clientid**))**

**{**

**return**;

**}**

//Generating a Uniqure Refresh Token ID

var refreshTokenId = Guid.NewGuid**()**.ToString**(**"n"**)**;

**using** **(**AuthenticationRepository \_repo = new AuthenticationRepository**())**

**{**

// Getting the Refesh Token Life Time From the Owin Context

var refreshTokenLifeTime = context.OwinContext.Get**<**string**>(**"ta:clientRefreshTokenLifeTime"**)**;

//Creating the Refresh Token object

var token = new RefreshToken**()**

**{**

//storing the RefreshTokenId in hash format

ID = Helper.GetHash**(**refreshTokenId**)**,

ClientID = clientid,

UserName = context.Ticket.Identity.Name,

IssuedTime = DateTime.UtcNow,

ExpiredTime = DateTime.UtcNow.AddMinutes**(**Convert.ToDouble**(**refreshTokenLifeTime**))**

**}**;

//Setting the Issued and Expired time of the Refresh Token

context.Ticket.Properties.IssuedUtc = token.IssuedTime;

context.Ticket.Properties.ExpiresUtc = token.ExpiredTime;

token.ProtectedTicket = context.SerializeTicket**()**;

var result = **await** \_repo.AddRefreshToken**(**token**)**;

**if** **(**result**)**

**{**

context.SetToken**(**refreshTokenId**)**;

**}**

**}**

**}**

**public** Task ReceiveAsync**(**AuthenticationTokenReceiveContext context**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **void** Create**(**AuthenticationTokenCreateContext context**)**

**{**

**throw** new NotImplementedException**()**;

**}**

**public** **void** Receive**(**AuthenticationTokenReceiveContext context**)**

**{**

**throw** new NotImplementedException**()**;

**}**

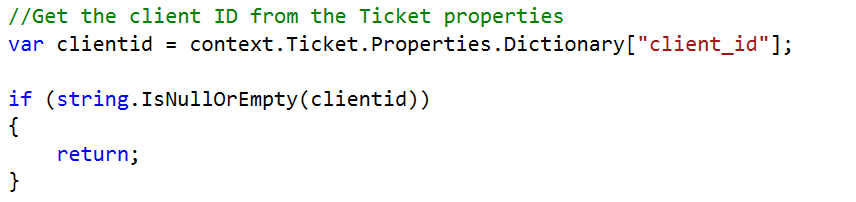
**}**

**}**

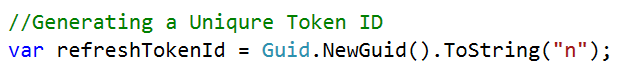
**Explanation of the above code:**

As shown above, the class RefreshTokenProvider implements the interface IAuthenticationTokenProvider, and here we need to add our refresh token generation logic inside the method CreateAsync.

Let discuss what we have done inside the CreateAsync method. First, we are getting the client id from the Ticket Properties. The following code does the above things.



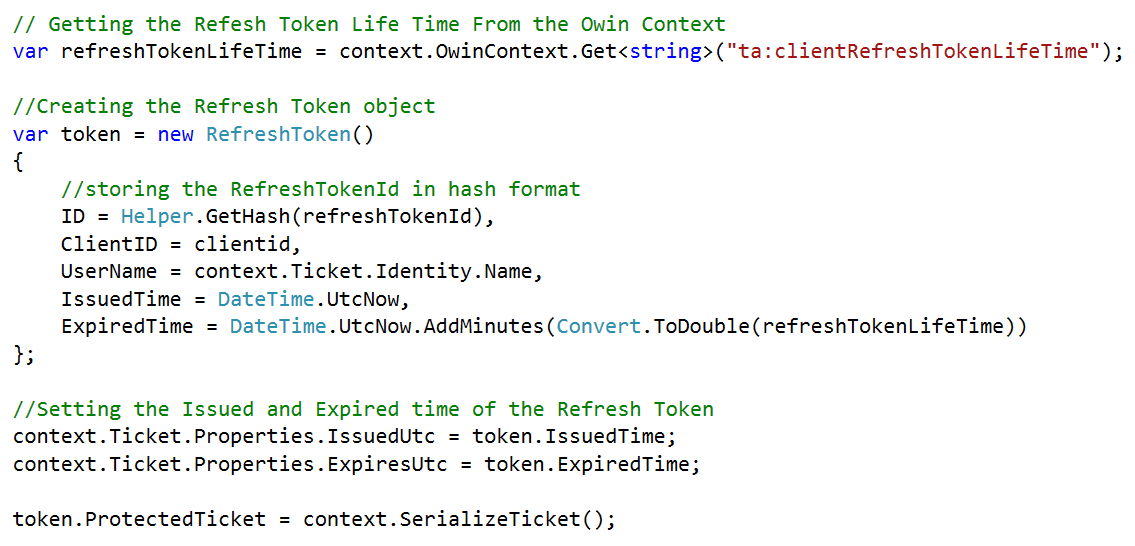
Next, we are generating a unique identifier for the refresh token, here, I am using Guid which is enough for this or you can use your own unique string generation algorithm. The following code exactly does the same.



Then we are reading the refresh token lifetime value from the Owin context and this value was set when we validate the client, this value will be used to determine how long the refresh token will be valid for, this should be in minutes.

Then we are setting the IssuedUtc, and ExpiresUtc values for the ticket, setting those properties will determine how long the refresh token in web api will be valid for.

After setting all context properties we are calling the context.SerializeTicket() method which will be responsible to serialize the ticket content and we will be able to store this magical serialized string on to the database. The following diagram shows the above things.



Now we strong the above token record into the **RefreshTokens** table, note that we are checking the token which will be saved on the database is unique for this Username (User) and the Client, if it not unique first we will delete the existing one and then store the new refresh token. It is better to hash the refresh token identifier before storing it, so if anyone has access to the database he will not be able to see the real refresh tokens.

Finally, we will send back the refresh token id (without hashing it) in the response body. The following does the above thing.

Refresh Token in Web API

**Step10: Modifying the Start class (OwinStartup class)**

We need to set the RefreshTokenProvider class within the OAuthAuthorizationServerOptions, so open the class Start which is present inside the app\_start folder and replace the code used to set OAuthAuthorizationServerOptions, with the below code, you can notice that we are setting the access token life time to a short period now (30 minutes) instead of 24 hours.

**using** *System;*

**using** *Microsoft.Owin;*

**using** *Owin;*

**using** *TokenAuthenticationInWebAPI.Models;*

**using** *Microsoft.Owin.Security.OAuth;*

**using** *System.Web.Http;*

**[**assembly: OwinStartup**(**typeof**(**TokenAuthenticationInWebAPI.App\_Start.Startup**))]**

**namespace** *TokenAuthenticationInWebAPI.App\_Start*

**{**

**public** **class** Startup

**{**

**public** **void** Configuration**(**IAppBuilder app**)**

**{**

OAuthAuthorizationServerOptions options = new OAuthAuthorizationServerOptions

**{**

AllowInsecureHttp = **true**,

//The Path For generating the Toekn

TokenEndpointPath = new PathString**(**"/token"**)**,

//Setting the Token Expired Time (30 minutes)

AccessTokenExpireTimeSpan = TimeSpan.FromMinutes**(**30**)**,

//MyAuthorizationServerProvider class will validate the user credentials

Provider = new MyAuthorizationServerProvider**()**,

//For creating the refresh token and regenerate the new access token

RefreshTokenProvider = new RefreshTokenProvider**()**

**}**;

app.UseOAuthAuthorizationServer**(**options**)**;

app.UseOAuthBearerAuthentication**(**new OAuthBearerAuthenticationOptions**())**;

HttpConfiguration config = new HttpConfiguration**()**;

WebApiConfig.Register**(**config**)**;

**}**

**}**

**}**

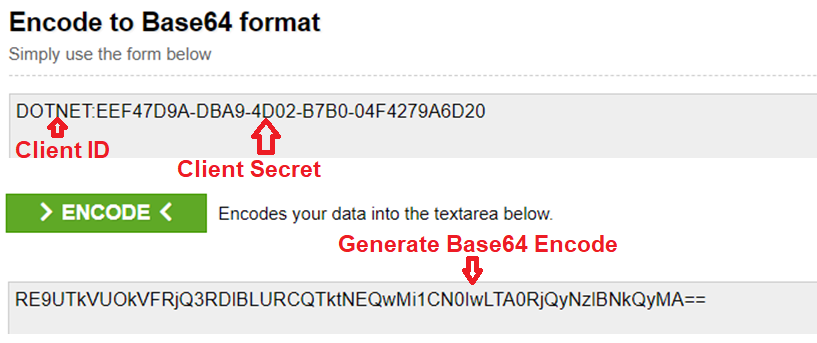
**Step11: Testing the API using Postman:**

Let’s first create the Base64 Encode value by for the ClientID and ClientSecret by using the following website

**<https://www.base64encode.org/>**

Enter the Client ID and Client Secret separated by a colon (:) in **“Encode to Base64 format”** textbox, and then click on the **“Encode”**button as shown in the below diagram which will generate the Base64 encoded value.

Example: **ClientID**: DOTNET and **Client** **Secret**: EEF47D9A-DBA9-4D02-B7B0-04F4279A6D20



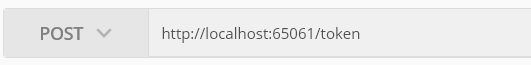
Base64 Code value: RE9UTkVUOkVFRjQ3RDlBLURCQTktNEQwMi1CN0IwLTA0RjQyNzlBNkQyMA==

Once you generate the Base64 encoded value, let’s see how to use basic authentication in the header to pass the Base64 encoded value. Here we need to use the Authorization header and the value will be the Base64 encoded string followed the “BASIC” as shown below.

**Authorization: BASIC**RE9UTkVUOkVFRjQ3RDlBLURCQTktNEQwMi1CN0IwLTA0RjQyNzlBNkQyMA==

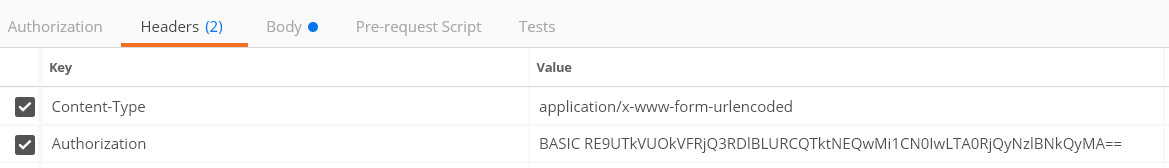
**Let’s see step by step procedure to use the Postman to generate the Access Token**

**Step1**: Select the Method as POST and provide URI as shown below in the below image

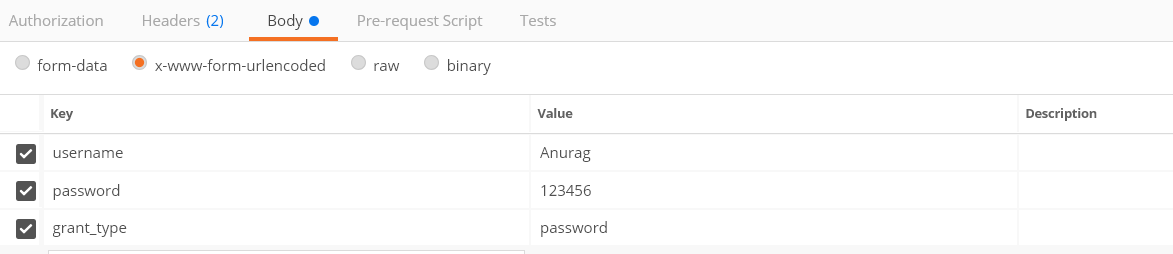


**Step2:**Select the Header tab and provide the Authorization value as shown below.

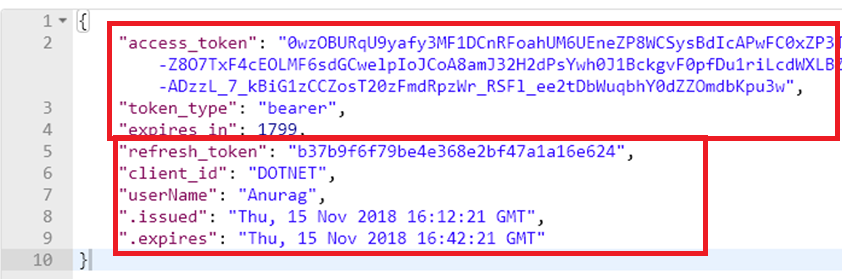
**Authorization: BASIC**RE9UTkVUOkVFRjQ3RDlBLURCQTktNEQwMi1CN0IwLTA0RjQyNzlBNkQyMA==



**Step3:**Select the Body Tab. Then choose x-www-form-urlencoded option and provide the username and password value. Provide the grant\_type value as password as shown in the below image.



Now click on the Send button which will generate the access token along with the refresh token as shown below.



As shown in the response body, you will notice that we have obtained a refresh\_token along with the access token which should be used to obtain a new access token (we will discuss this after a while in this post) this token is bounded to the user Anurag and for the Client DOTNET. Note that the expires\_in value is related to the access token, not the refresh token, this access token will expire in 30 mins.

**Step12: Generating an Access Token using the Refresh Token in Web API**

Now we need to implement the logic needed to generate a new access token when we receive the request from the refresh the token, to do so open the class RefreshTokenProvider and implement the ReceiveAsync method as shown below.

**public** **async** Task ReceiveAsync**(**AuthenticationTokenReceiveContext context**)**

**{**

var allowedOrigin = context.OwinContext.Get**<**string**>(**"ta:clientAllowedOrigin"**)**;

context.OwinContext.Response.Headers.Add**(**"Access-Control-Allow-Origin", new**[]** **{** allowedOrigin **})**;

string hashedTokenId = Helper.GetHash**(**context.Token**)**;

**using** **(**AuthenticationRepository \_repo = new AuthenticationRepository**())**

**{**

var refreshToken = **await** \_repo.FindRefreshToken**(**hashedTokenId**)**;

**if** **(**refreshToken != **null)**

**{**

//Get protectedTicket from refreshToken class

context.DeserializeTicket**(**refreshToken.ProtectedTicket**)**;

var result = **await** \_repo.RemoveRefreshTokenByID**(**hashedTokenId**)**;

**}**

**}**

**}**

**Explanation of the above method:**

We need to set the “Access-Control-Allow-Origin” header by getting the value from the Owin Context. If you will not set this value, then you will get 405 status code and this is because the method “GrantResourceOwnerCredentials” where we set this header is never get executed once we request the access token using the refresh tokens (grant\_type = refresh\_token).

Then we get the refresh token id from the request, hash this id and look for the token using the hashed refresh token id in “RefreshToken” table, if the refresh token is found, we will use the magical signed string which contains a serialized representation for the ticket to build the ticket and identities for the user mapped to this refresh token.

Finally, we will remove the existing refresh token from the “RefreshToken” table because in our logic we are allowing only one refresh token per user and client.

**Implementing GrantRefreshToken in Web API**

Now the request context contains all the claims stored previously for this use. Now we need to implement the logic which will allows us to issue new claims or updating the existing claims and contain them into the new access token generated before sending it to the user, to do so open class MyAuthorizationServerProvider and implement method GrantRefreshToken with the following code.

**public** **override** Task GrantRefreshToken**(**OAuthGrantRefreshTokenContext context**)**

**{**

var originalClient = context.Ticket.Properties.Dictionary**[**"client\_id"**]**;

var currentClient = context.ClientId;

**if** **(**originalClient != currentClient**)**

**{**

context.SetError**(**"invalid\_clientId", "Refresh token is issued to a different clientId."**)**;

**return** Task.FromResult**<object>(null)**;

**}**

// Change auth ticket for refresh token requests

var newIdentity = new ClaimsIdentity**(**context.Ticket.Identity**)**;

newIdentity.AddClaim**(**new Claim**(**"newClaim", "newValue"**))**;

var newTicket = new AuthenticationTicket**(**newIdentity, context.Ticket.Properties**)**;

context.Validated**(**newTicket**)**;

**return** Task.FromResult**<object>(null)**;

**}**

**Explanation of the above code:**

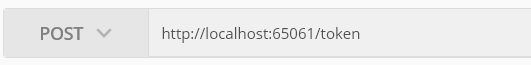
First, we are reading the client id value from the original ticket and this is the client ids which get stored in the magical signed string. Then we are comparing this client id with the client id sent with the request, if they are different then we will reject this request because we need to make sure that the refresh token used here is bound to the same client when it was generated.

Now, we have the chance to add new claims or remove or update existing claims, to do this we are calling the “context.Validated(newTicket)” method which will generate the new access token and return it in the response body.

Lastly, after this method executes successfully, the flow for the code will hit the “CreateAsync” method which is present in the class “RefreshTokenProvider” and a new refresh token is generated and returned in the response along with the new access token.

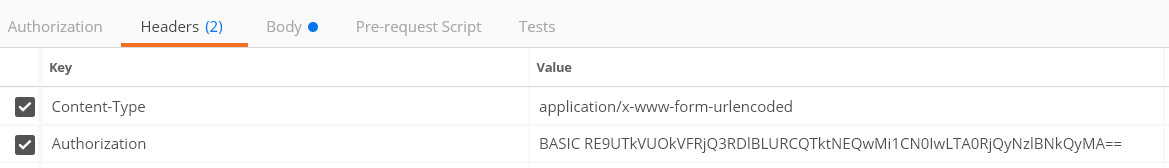
**Testing the Refresh Token in Web API with Postman to generate new access Token:**

**Step1**: Select the Method as POST and provide URI as shown below in the below image

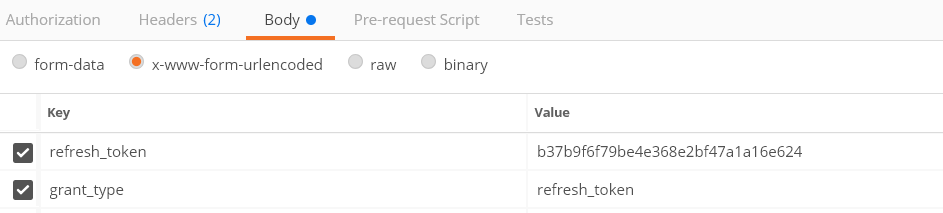


**Step2:**Select the Header tab and provide the Authorization value as shown below. This is the Base64 encoded value for the ClientID and Client Secret.

**Authorization: BASIC**RE9UTkVUOkVFRjQ3RDlBLURCQTktNEQwMi1CN0IwLTA0RjQyNzlBNkQyMA==



**Step3:**Select the Body Tab. Then choose x-www-form-urlencoded option and provide the Refresh\_Token value and the grant\_type value as refresh\_token as shown in the below image.



**Consume Refresh Token in C# Client**

In this article, I will discuss how to **Consume Refresh Token in C#** **Client** application. Please read the following three articles, before proceeding to this article as we are going to consume the services that we created in our previous articles.

**[Token Based Authentication in Web API](https://dotnettutorials.net/lesson/token-based-authentication-in-web-api/)**: In this article, we discussed how to implement and use the Token Based Authentication in Web API.

**[Client Validation in Token Based Authentication](https://dotnettutorials.net/lesson/client-validation-using-basic-authentication-web-api/)**: In this article, we discussed how to validate the clients while generating the token in Web API.

**[Generating Refresh Token in Web API](https://dotnettutorials.net/lesson/refresh-token-in-web-api/)**: In this article, we discussed how to Generate Refresh Token in Web API.

Let us discuss the step by step procedure to Consume Refresh Token in C#. But before that let’s modify the Test Controller of our Web API application that we created in our previous application as shown below.

**Step1: Modifying the Test Controller**

**using** *System.Linq;*

**using** *System.Security.Claims;*

**using** *System.Web.Http;*

**namespace** *TokenAuthenticationInWebAPI.Controllers*

**{**

**public** **class** TestController : ApiController

**{**

//This resource is For all types of role

**[**Authorize**(**Roles = "SuperAdmin, Admin, User"**)]**

**[**HttpGet**]**

**[**Route**(**"api/test/resource1"**)]**

**public** IHttpActionResult GetResource1**()**

**{**

var identity = **(**ClaimsIdentity**)**User.Identity;

**return** Ok**(**"Hello: " + identity.Name**)**;

**}**

//This resource is only For Admin and SuperAdmin role

**[**Authorize**(**Roles = "SuperAdmin, Admin"**)]**

**[**HttpGet**]**

**[**Route**(**"api/test/resource2"**)]**

**public** IHttpActionResult GetResource2**()**

**{**

var identity = **(**ClaimsIdentity**)**User.Identity;

var Email = identity.Claims

.FirstOrDefault**(**c =**>** c.Type == "Email"**)**.Value;

var UserName = identity.Name;

**return** Ok**(**"Hello " + UserName + ", Your Email ID is :" + Email**)**;

**}**

//This resource is only For SuperAdmin role

**[**Authorize**(**Roles = "SuperAdmin"**)]**

**[**HttpGet**]**

**[**Route**(**"api/test/resource3"**)]**

**public** IHttpActionResult GetResource3**()**

**{**

var identity = **(**ClaimsIdentity**)**User.Identity;

var roles = identity.Claims

.Where**(**c =**>** c.Type == ClaimTypes.Role**)**

.Select**(**c =**>** c.Value**)**;

**return** Ok**(**"Hello " + identity.Name + "Your Role(s) are: " + string.Join**(**",", roles.ToList**()))**;

**}**

**}**

**}**

**Step2: Creating the UserTokenMaster table**

In the client side, we need to store the token in the UserTokenMaster table as shown below

Consume Refresh Token in C#

**Please use the below SQL Script to create the required database.**

**CREATE** **DATABASE** Test\_DB

**GO**

**USE** Test\_DB

**CREATE** **TABLE** UserTokenMaster

(

UserName **VARCHAR**(50) **PRIMARY** **KEY**,

UserPassword **VARCHAR**(50),

AccessToken **VARCHAR**(500),

RefreshToken **VARCHAR**(100),

TokenExpiredTime **DATETIME**

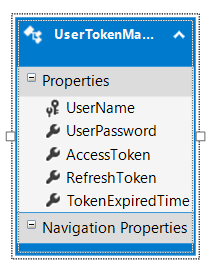
)

**GO**

**Step3: Create a new console application with the name RefreshTokenClient1.**

**Step4: Add ADO.NET Entity Data Model**

Here, we need to add ADO.NET Entity Data Model Database First approach against the Test\_DB and add the UserTokenMaster to the EDMX that we created in Step2. So once you add the table, the EDMX File should be as shown below



**Step5: Add Token class**

Now we need to add a class file with the name Token to the project. And then copy and paste the following code.

**using** *Newtonsoft.Json;*

**using** *System;*

**namespace** *RefreshTokenClient1*

**{**

// The body of the response from API is a JSON object that

// contains the following properties (and a couple of others

// that we're not capturing).

**public** **class** Token

**{**

**[**JsonProperty**(**"access\_token"**)]**

**public** string AccessToken **{** **get**; **set**; **}**

**[**JsonProperty**(**"token\_type"**)]**

**public** string TokenType **{** **get**; **set**; **}**

**[**JsonProperty**(**"expires\_in"**)]**

**public** **int** ExpiresIn **{** **get**; **set**; **}**

**[**JsonProperty**(**"refresh\_token"**)]**

**public** string RefreshToken **{** **get**; **set**; **}**

**public** string Error **{** **get**; **set**; **}**

**public** DateTime ExpiredDateTime **{** **get**; **set**; **}**

**}**

**}**

**Step6: Adding UserTokenRepository**

Now we need to add a class with the name UserTokenRepository and within that class we are going to perform the database operations. So once you add the class, copy and paste the following code.

**using** *System;*

**using** *System.Linq;*

**namespace** *RefreshTokenClient1*

**{**

**class** UserTokenRepository : IDisposable

**{**

// Test\_DBEntities it is your context class

Test\_DBEntities context = new Test\_DBEntities**()**;

**public** Token GetTokenFromDB**(**string username, string password**)**

**{**

UserTokenMaster userMaster = context.UserTokenMasters.FirstOrDefault**(**user =**>**

user.UserName.Equals**(**username, StringComparison.OrdinalIgnoreCase**)**

&& user.UserPassword == password**)**;

Token token = **null**;

**if** **(**userMaster != **null)**

**{**

token = new Token**()**

**{**

AccessToken = userMaster.AccessToken,

RefreshToken = userMaster.RefreshToken,

ExpiredDateTime = **(**DateTime**)**userMaster.TokenExpiredTime

**}**;

**}**

**return** token;

**}**

//Adding Token into the DB

**public** **bool** AddUserTokenIntoDB**(**UserTokenMaster userTokenMaster**)**

**{**

//First Check the existance of the Token in the DB

var tokenMaster = context.UserTokenMasters.FirstOrDefault**(**x =**>** x.UserName == userTokenMaster.UserName

&& x.UserPassword == userTokenMaster.UserPassword**)**;

**if** **(**tokenMaster != **null)**

**{**

context.UserTokenMasters.Remove**(**tokenMaster**)**;

**}**

context.UserTokenMasters.Add**(**userTokenMaster**)**;

**bool** isAdded = context.SaveChanges**()** **>** 0;

**return** isAdded;

**}**

**public** **void** Dispose**()**

**{**

context.Dispose**()**;

**}**

**}**

**}**

**Step7: Modify the Program class**

Here we need to implement the logic to get the access token and refresh from token API and then storing the Token into our database.

**Note:** The editor we are using to display the code snippet is not showing the following code, so we display the code as it is. The following code is self-explained, please go through the comments.

using System;  
using System.Collections.Generic;  
using System.Net.Http;  
using System.Net.Http.Headers;  
using System.Text;  
using Newtonsoft.Json;  
namespace RefreshTokenClient1  
{  
class Program  
{  
// When your application is registered you will get  
// the client id and secret from the API  
private static string \_clientId = “DOTNET”;  
private static string \_clientSecret = “EEF47D9A-DBA9-4D02-B7B0-04F4279A6D20”;  
private static string Username = “”;  
private static string Password = “”;  
//Store the base address of the web api  
//You need to change the PORT number where your WEB API service is running  
private static string baseAddress = “http://localhost:65061/”;  
static void Main(string[] args)  
{  
Token token = null;  
Username = “Anurag”;  
Password = “123456”;  
// First get the token from the persistent storage based  
// on the username and password  
token = (new UserTokenRepository()).GetTokenFromDB(Username, Password);  
//Then check the existing token and its expiration datetime  
if (token != null && DateTime.Now < token.ExpiredDateTime)  
{  
//use the existing token  
}  
else if (token != null && !string.IsNullOrEmpty(token.RefreshToken))  
{  
//Get a new access token based on the Refresh Token  
token = GetTokens(\_clientId, \_clientSecret, token.RefreshToken);  
}  
else  
{  
//Get a brand new access token  
token = GetTokens(\_clientId, \_clientSecret, null);  
}  
//If you get the access token, then call the authorized resource  
if (!string.IsNullOrEmpty(token.AccessToken))  
{  
CallAPIResource1(token.AccessToken);  
}  
else  
{  
Console.WriteLine(token.Error);  
}  
Console.ReadLine();  
}  
//Here we implment the logic to call the authorized resource  
private static void CallAPIResource1(string AccessToken)  
{  
HttpClientHandler handler = new HttpClientHandler();  
HttpClient client = new HttpClient(handler);  
// Need to set the Access Token in the Authorization Header as shown below  
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue(“Bearer”, AccessToken);  
// Make a Get request for the authorized resource by invoking  
// the PostAsync method on the client object as shown below  
var APIResponse = client.GetAsync(baseAddress + “api/test/resource1”).Result;  
if (APIResponse.IsSuccessStatusCode)  
{  
var JsonContent = APIResponse.Content.ReadAsStringAsync().Result;  
string Message = JsonConvert.DeserializeObject<string>(JsonContent);  
Console.WriteLine(“APIResponse : ” + Message);  
}  
else  
{  
Console.WriteLine(“APIResponse, Error : ” + APIResponse.StatusCode);  
}  
}  
//In this method we need to implement the logic whether we need get a brand new access token  
// or we need the access token based on the Refresh Token  
private static Token GetTokens(string clientId, string clientSecret, string RefreshToken)  
{  
Token token = null;  
if (string.IsNullOrEmpty(RefreshToken))  
{  
token = GetAccessToken(clientId, clientSecret, Username, Password);  
}  
else  
{  
token = GetAccessTokenByRefreshToken(clientId, clientSecret, RefreshToken);  
// The Refresh token can become invalid for several reasons  
// such as invalid cliendid and secret or the user’s password has changed.  
// In Such cases issue a brand new access token  
if (!string.IsNullOrEmpty(token.Error))  
{  
token = GetAccessToken(clientId, clientSecret, Username, Password);  
}  
}  
if (!string.IsNullOrEmpty(token.Error))  
{  
throw new Exception(token.Error);  
}  
else  
{  
//Need to store the token in any presistent storage  
token.ExpiredDateTime = DateTime.Now.AddSeconds(token.ExpiresIn);  
//Create an object of type UserTokenMaster  
UserTokenMaster userTokenMaster = new UserTokenMaster()  
{  
UserName = Username,  
UserPassword = Password,  
AccessToken = token.AccessToken,  
RefreshToken = token.RefreshToken,  
TokenExpiredTime = token.ExpiredDateTime  
};  
bool IsAddeded = (new UserTokenRepository()).AddUserTokenIntoDB(userTokenMaster);  
if (IsAddeded)  
{  
token.Error = “Error Occurred while saving the Token into the DB”;  
}  
}  
return token;  
}  
//This method is used to get a new access token  
public static Token GetAccessToken(string clientId, string clientSecret, string username, string password)  
{  
Token token = new Token();  
HttpClientHandler handler = new HttpClientHandler();  
HttpClient client = new HttpClient(handler);  
// Need to set the Client ID and Client Secret in the Authorization Header  
// in Base64 Encoded Format using the Basic Authentication as shown below  
string ClientIDandSecret = clientId + “:” + clientSecret;  
var authorizationHeader = Convert.ToBase64String(Encoding.UTF8.GetBytes(ClientIDandSecret));  
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue(“Basic”, authorizationHeader);  
// Create a dictionary which contains the request form data, here we need to set  
// the username, password and grant\_type as shown below  
var RequestBody = new Dictionary<string, string>  
{  
{“grant\_type”, “password”},  
{“username”, username},  
{“password”, password},  
};  
//Make a Post request by invoking the PostAsync method on the client object as shown below  
var tokenResponse = client.PostAsync(baseAddress + “token”, new FormUrlEncodedContent(RequestBody)).Result;  
if (tokenResponse.IsSuccessStatusCode)  
{  
var JsonContent = tokenResponse.Content.ReadAsStringAsync().Result;  
token = JsonConvert.DeserializeObject<Token>(JsonContent);  
token.Error = null;  
}  
else  
{  
token.Error = “GetAccessToken failed likely due to an invalid client ID, client secret, or invalid usrename and password”;  
}  
return token;  
}  
//This method is used to get a new access token based on the Refresh Token  
public static Token GetAccessTokenByRefreshToken(string clientId, string clientSecret, string refreshToken)  
{  
Token token = new Token();  
HttpClientHandler handler = new HttpClientHandler();  
HttpClient client = new HttpClient(handler);  
// Need to set the Client ID and Client Secret in the Authorization Header  
// in Base64 Encoded Format using Basic Authentication as shown below  
string ClientIDandSecret = clientId + “:” + clientSecret;  
var authorizationHeader = Convert.ToBase64String(Encoding.UTF8.GetBytes(ClientIDandSecret));  
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue(“Basic”, authorizationHeader);  
// Create a dictionary which contains the refresh token, here we need to set  
// the grant\_type as refresh\_token as shown below  
var RequestBody = new Dictionary<string, string>  
{  
{“grant\_type”, “refresh\_token”},  
{“refresh\_token”, refreshToken}  
};  
//Make a Post request by invoking the PostAsync method on the client object as shown below  
var tokenResponse = client.PostAsync(baseAddress + “token”, new FormUrlEncodedContent(RequestBody)).Result;  
if (tokenResponse.IsSuccessStatusCode)  
{  
var JsonContent = tokenResponse.Content.ReadAsStringAsync().Result;  
token = JsonConvert.DeserializeObject<Token>(JsonContent);  
token.Error = null;  
}  
else  
{  
token.Error = “GetAccessToken by Refresh Token failed likely due to an invalid client ID, client secret, or it has been revoked by the system admin”;  
}  
return token;  
}  
}  
}

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Net.Http;*

**using** *System.Net.Http.Headers;*

**using** *System.Text;*

**using** *Newtonsoft.Json;*

**namespace** *RefreshTokenClient1*

**{**

**class** Program

**{**

// When your application is registered you will get

// the client id and secret from the API

**private** **static** string \_clientId = "DOTNET";

**private** **static** string \_clientSecret = "EEF47D9A-DBA9-4D02-B7B0-04F4279A6D20";

**private** **static** string Username = "";

**private** **static** string Password = "";

//Store the base address of the web api

//You need to change the PORT number where your WEB API service is running

**private** **static** string baseAddress = "http://localhost:65061/";

**static** **void** Main**(**string**[]** args**)**

**{**

Token token = **null**;

Username = "Anurag";

Password = "123456";

// First get the token from the persistent storage based

// on the username and password

token = **(**new UserTokenRepository**())**.GetTokenFromDB**(**Username, Password**)**;

//Then check the existing token and its expiration datetime

**if** **(**token != **null** && DateTime.Now **<** token.ExpiredDateTime**)**

**{**

//use the existing token

**}**

**else** **if** **(**token != **null** && !string.IsNullOrEmpty**(**token.RefreshToken**))**

**{**

//Get a new access token based on the Refresh Token

token = GetTokens**(**\_clientId, \_clientSecret, token.RefreshToken**)**;

**}**

**else**

**{**

//Get a brand new access token

token = GetTokens**(**\_clientId, \_clientSecret, **null)**;

**}**

//If you get the access token, then call the authorized resource

**if** **(**!string.IsNullOrEmpty**(**token.AccessToken**))**

**{**

CallAPIResource1**(**token.AccessToken**)**;

**}**

**else**

**{**

Console.WriteLine**(**token.Error**)**;

**}**

Console.ReadLine**()**;

**}**

//Here we implment the logic to call the authorized resource

**private** **static** **void** CallAPIResource1**(**string AccessToken**)**

**{**

HttpClientHandler handler = new HttpClientHandler**()**;

HttpClient client = new HttpClient**(**handler**)**;

// Need to set the Access Token in the Authorization Header as shown below

client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue**(**"Bearer", AccessToken**)**;

// Make a Get request for the authorized resource by invoking

// the PostAsync method on the client object as shown below

var APIResponse = client.GetAsync**(**baseAddress + "api/test/resource1"**)**.Result;

**if** **(**APIResponse.IsSuccessStatusCode**)**

**{**

var JsonContent = APIResponse.Content.ReadAsStringAsync**()**.Result;

string Message = JsonConvert.DeserializeObject**<**string**>(**JsonContent**)**;

Console.WriteLine**(**"APIResponse : " + Message**)**;

**}**

**else**

**{**

Console.WriteLine**(**"APIResponse, Error : " + APIResponse.StatusCode**)**;

**}**

**}**

//In this method we need to implement the logic whether we need get a brand new access token

// or we need the access token based on the Refresh Token

**private** **static** Token GetTokens**(**string clientId, string clientSecret, string RefreshToken**)**

**{**

Token token = **null**;

**if** **(**string.IsNullOrEmpty**(**RefreshToken**))**

**{**

token = GetAccessToken**(**clientId, clientSecret, Username, Password**)**;

**}**

**else**

**{**

token = GetAccessTokenByRefreshToken**(**clientId, clientSecret, RefreshToken**)**;

// The Refresh token can become invalid for several reasons

// such as invalid cliendid and secret or the user's password has changed.

// In Such cases issue a brand new access token

**if** **(**!string.IsNullOrEmpty**(**token.Error**))**

**{**

token = GetAccessToken**(**clientId, clientSecret, Username, Password**)**;

**}**

**}**

**if** **(**!string.IsNullOrEmpty**(**token.Error**))**

**{**

**throw** new Exception**(**token.Error**)**;

**}**

**else**

**{**

//Need to store the token in any presistent storage

token.ExpiredDateTime = DateTime.Now.AddSeconds**(**token.ExpiresIn**)**;

//Create an object of type UserTokenMaster

UserTokenMaster userTokenMaster = new UserTokenMaster**()**

**{**

UserName = Username,

UserPassword = Password,

AccessToken = token.AccessToken,

RefreshToken = token.RefreshToken,

TokenExpiredTime = token.ExpiredDateTime

**}**;

**bool** IsAddeded = **(**new UserTokenRepository**())**.AddUserTokenIntoDB**(**userTokenMaster**)**;

**if** **(**IsAddeded**)**

**{**

token.Error = "Error Occurred while saving the Token into the DB";

**}**

**}**

**return** token;

**}**

//This method is used to get a new access token

**public** **static** Token GetAccessToken**(**string clientId, string clientSecret, string username, string password**)**

**{**

Token token = new Token**()**;

HttpClientHandler handler = new HttpClientHandler**()**;

HttpClient client = new HttpClient**(**handler**)**;

// Need to set the Client ID and Client Secret in the Authorization Header

// in Base64 Encoded Format using the Basic Authentication as shown below

string ClientIDandSecret = clientId + ":" + clientSecret;

var authorizationHeader = Convert.ToBase64String**(**Encoding.UTF8.GetBytes**(**ClientIDandSecret**))**;

client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue**(**"Basic", authorizationHeader**)**;

// Create a dictionary which contains the request form data, here we need to set

// the username, password and grant\_type as shown below

var RequestBody = new Dictionary**<**string, string**>**

**{**

**{**"grant\_type", "password"**}**,

**{**"username", username**}**,

**{**"password", password**}**,

**}**;

//Make a Post request by invoking the PostAsync method on the client object as shown below

var tokenResponse = client.PostAsync**(**baseAddress + "token", new FormUrlEncodedContent**(**RequestBody**))**.Result;

**if** **(**tokenResponse.IsSuccessStatusCode**)**

**{**

var JsonContent = tokenResponse.Content.ReadAsStringAsync**()**.Result;

token = JsonConvert.DeserializeObject**<**Token**>(**JsonContent**)**;

token.Error = **null**;

**}**

**else**

**{**

token.Error = "GetAccessToken failed likely due to an invalid client ID, client secret, or invalid usrename and password";

**}**

**return** token;

**}**

//This method is used to get a new access token based on the Refresh Token

**public** **static** Token GetAccessTokenByRefreshToken**(**string clientId, string clientSecret, string refreshToken**)**

**{**

Token token = new Token**()**;

HttpClientHandler handler = new HttpClientHandler**()**;

HttpClient client = new HttpClient**(**handler**)**;

// Need to set the Client ID and Client Secret in the Authorization Header

// in Base64 Encoded Format using Basic Authentication as shown below

string ClientIDandSecret = clientId + ":" + clientSecret;

var authorizationHeader = Convert.ToBase64String**(**Encoding.UTF8.GetBytes**(**ClientIDandSecret**))**;

client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue**(**"Basic", authorizationHeader**)**;

// Create a dictionary which contains the refresh token, here we need to set

// the grant\_type as refresh\_token as shown below

var RequestBody = new Dictionary**<**string, string**>**

**{**

**{**"grant\_type", "refresh\_token"**}**,

**{**"refresh\_token", refreshToken**}**

**}**;

//Make a Post request by invoking the PostAsync method on the client object as shown below

var tokenResponse = client.PostAsync**(**baseAddress + "token", new FormUrlEncodedContent**(**RequestBody**))**.Result;

**if** **(**tokenResponse.IsSuccessStatusCode**)**

**{**

var JsonContent = tokenResponse.Content.ReadAsStringAsync**()**.Result;

token = JsonConvert.DeserializeObject**<**Token**>(**JsonContent**)**;

token.Error = **null**;

**}**

**else**

**{**

token.Error = "GetAccessToken by Refresh Token failed likely due to an invalid client ID, client secret, or it has been revoked by the system admin";

**}**

**return** token;

**}**

**}**

**}**

That’s it. Run the application and see everything is working as expected. In the next article, I will discuss how to consume Refresh Token using JavaScript.

**HMAC Authentication in Web API**

In this article, I am going to discuss how to implement the **HMAC Authentication in Web API**Application**.**Please read our previous article where we discussed **[Token Based Authentication in Web API](https://dotnettutorials.net/lesson/token-based-authentication-web-api/)**. The most important thing that you need to be considered while developing API is to ensure its security as the API will be exposed over the network and HMAC Authentication is one of the mechanisms to provide security to the Web API Resources. As part of this article, we are going to discuss the following pointers in detail.

1. **What is HMAC Authentication?**
2. **Understanding the Keys used in HMAC Authentication.**
3. **Uses of HMAC Authentication in Web API.**
4. **How does the HMAC Authentication work?**  
   **The flow of HMAC on the Client Side.**  
   **The flow of HMAC on the server-side.**
5. **Implementing HAMC in both client and server.**
6. **Understanding the Replay Request?**

**What is HMAC Authentication?**

The **HMAC** stands for **Hash-based Message Authentication Code**. From the full form of **HMAC**, we need to understand two things one is **Message Authentication Code** and the other one is **Hash-Based.** So HMAC is a mechanism which is used for creating a **Message Authentication Code** by using a **Hash Function**.

The most important thing that we need to keep in mind is that while generating the **Message Authentication Code using Hash Function** we need to use a Shared **Secret Key**. Moreover that Shared **Secret Key** must be shared between the Client and the Server involved in sending and receiving the data.

**What are the Keys used in HMAC Authentication in Web API?**

First of all, the server needs to generate**two keys** one is **Public Shared APP ID** and the other one is **Private Secret API Key.** Once the keys are generated then it is the responsibility of the Server to provide these keys to the Client using a secure channel like email and this should be done only once and that is too when the client registers with the server.

Once the Client get the keys, then it is the responsibility of the client to generate a unique **HMAC signature** (you can also say hash) which not only contain the request data but also contains all the necessary information which are required by the server to process the request and then the client send it to the server.

**NOTE:**

Usually, we need to creates the **HMAC Signature** (hash) by combining the request data. The Request Data contains the **Public APP Id**, **request URI**, **request content**, **HTTP method type**, **timestamp**, and **nonce** by using the **Private Secret API Key** (this key is not going to be sent in the request).

Once the server receives the request, then it tries to generate the hash (unique HMAC Signature) by using the data received from the client request. While the Server Generating the hash, it needs to use the same **Private Secret API Key** (which is used by the client) which was initially shared between the client and the server.

Once the hash (**unique HMAC Signature**) is generated by the server, then it is going to compare with the hash received from the client. If both the hashes are matched then the server will consider this request as a valid request and proceed else it simply returns unauthorized.

**Uses of HMAC Authentication in Web API**

The main uses of HMAC Authentication in Web API are as follows.

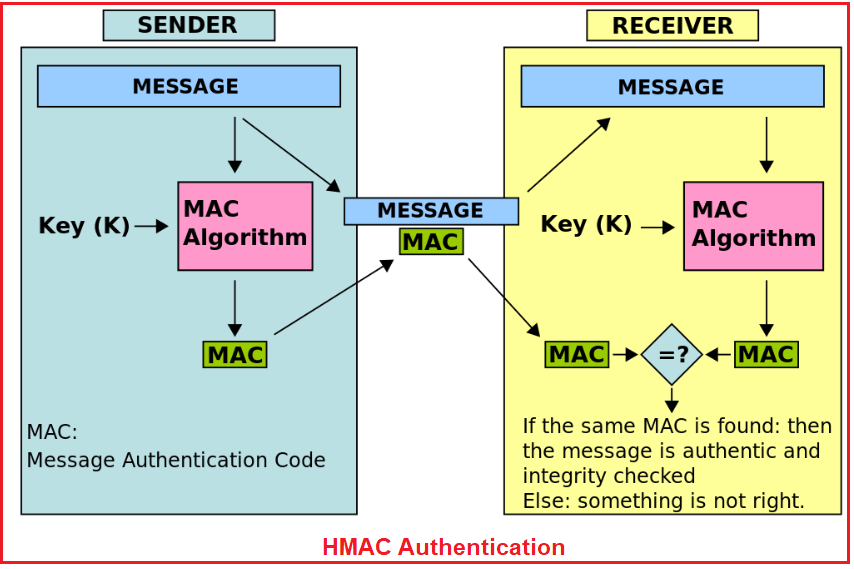
1. **Data integrity:** It means the data sent by the client to the server has not tampered.
2. **Request origination:** The request comes to the server from a trusted client.
3. **Not a replay request:** The request is not captured by an intruder and being replayed.

If this is not clear at the moment, then don’t worry we will discuss all the above points with real-time example after some time.

**How does the HMAC Authentication work?**

As of now, we have discussed the basic of **HMAC Authentication in Web API** from both the client and server point of view. Now let’s discuss the flow of Client and Server in details i.e. how the HMAC authentication works.

As we already discussed, first of all, the server should create and provide the two keys (**Public Shared APP Id** and **Private Secret API Key**) to the client. It is the responsibility of the client not to share the **Private Secret API Key** with anyone and moreover, the Client needs to store the **Private Secret API Key** securely, mostly in a database or in the config file.



Let first discuss the flow of Client in HMAC Authentication.

**The HMAC Flow on the Client Side:**

First, the client needs to create a string (MAC – Message Authentication Code) which will contain all the request data that the client wants to send to the server. Generally, the string contains the following parameters

1. **HTTP method**
2. **APP Id**
3. **Nonce**
4. **The request URI**
5. **Request timestamp**
6. **Base 64 string representation of the request payload** **(request body)**

**Note:**

Here, we need to calculate the**Request Time Stamp** value by using the **UNIX** time (number of seconds since Jan. 1st 1970). We need to do this to overcome the possibility of different time zone issues between the client and the server.

The **Nonce** is a random number or string which is used only once per request. Here we are going to use **GUID** to create Nonce.

Once the string is generated by combining all the parameters, then it is the responsibility of the client to generate a **HASH (unique signature)** of the above string by using any hashing algorithm such as **SHA256**. The important thing here you need to remember is that while generating the unique signature (hash), you need to use the **Private Secret API Key** which was initially provided by the server.

Once the **unique signature (hash)** is generated by the client, then the client needs to send that signature (hash) in the request header using a custom scheme such as “**hmacauth**”.

Here you can use any header but for the simplicity of this demo, we are going to use the **Authorization header** but it is not mandatory you can use any header.

The data in the header will contain the **public shared APP Id**, the **request time stamp**, and the **nonce** separated by a colon ‘:’. As we are going to use the **Authorization header**, so the format for the Authorization header should be as shown below:

**[Authorization: hmacauth APPId:Signature:Nonce:Timestamp]**

**The Flow of HMAC on the server-side:**

**Step1:**

The Server receives the request which contains the request data and the Authorization header. The Authorization header contains the HAMC signature. From the Authorization header, the server needs to extracts the values such as **APP Id, Signature, Nonce**and**Request Timestamp**.

**Step2:**

Once the server extracts the values from the Authorization header, then by using the **APP ID**(this value we will get in Step1)**,** the Server tries to get the **Private Secret API Key** which is generally stored in some secure repository such as a database or in the configuration file.

**Step3:**

Once the server gets the **Private Secret API Key**, then it tries to rebuild the string by combining the received request content data along with the extracted data (such as **APP Id, Nonce**and**Request Timestamp which are extracted in Step1**). The important thing here we need to understand is that the parameters order and the encoding format should be same as followed by the client.

**Step4:**

Once the string is built, then it is the responsibility of the Server to generate a hash value by using the same Hashing algorithm used by the client i.e. SHA256 using the Private Secret API Key which is already retrieved by the server in Step2.

**Step5:**

Once the hash is generated on the server, then the server compares the generated hash with the hash sent by the client, if both the hashes are matched then the server will consider this request as a valid request and process that request else it simply returns 401 unauthorized.

As of now, we have discussed lots of theory, you may have some doubts. Let’s clear your doubt with the practical implementation of the above theory. I divided the implementation into three sections.

1. **Section1: Generating the public APP ID and Shared Private API Key**
2. **Section2: Building the Client Application**
3. **Section3: Building the Server (Backend API)**

**Section1: Generating the public APP ID and Shared Private API Key**

Let’s first discuss how to generate the **Public Shared APP ID** and a strong 256 bits key which will act as our **Private Secret API Key**. This is usually done on the server and then provided to the client using a secure mechanism as we already discussed.

Here, we will use the **Symmetric Key Cryptographic Algorithm** to generate the 256 bits key which will be our **Private Secret API Key** and **GUID** to generate the **Public Shared APP ID**. To do so, create a console application and modify the Main method of the Program class as shown below to generate the**APP ID** and **API Key**:

**using** *System;*

**using** *System.Security.Cryptography;*

**namespace** *GeneratedClientAppIDAPPKey*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**using** **(**var cryptoProvider = new RNGCryptoServiceProvider**())**

**{**

var APPID = Guid.NewGuid**()**;

**byte[]** secretKeyByteArray = new **byte[**32**]**; //256 bit

cryptoProvider.GetBytes**(**secretKeyByteArray**)**;

var APIKey = Convert.ToBase64String**(**secretKeyByteArray**)**;

**}**

**}**

**}**

**}**

So for this demo let’s assume that our **APP ID** is **65d3a4f0-0239-404c-8394-21b94ff50604** and our **API Key** is **WLUEWeL3so2hdHhHM5ZYnvzsOUBzSGH4+T3EgrQ91KI=** and we also assume that we have provided these two Keys to our client using a secure channel.

**Section2: Building the Client Application**

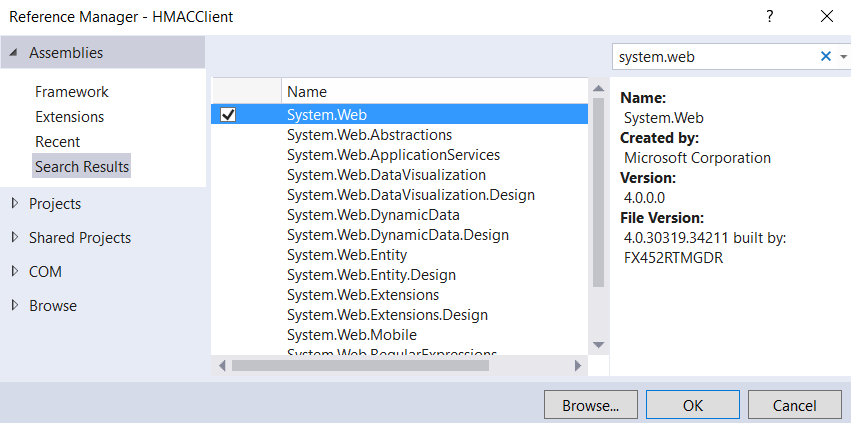
In this section, we are going to create the client application (Console Application) which will use HMAC Authentication. Let’s discuss the step by step process to achieve this.

**Step1: Creating a Console Application and installing the necessary Packages from NuGet**

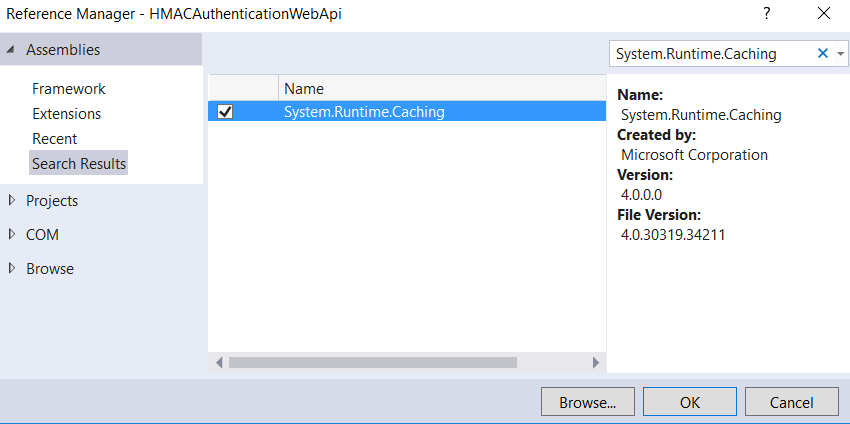
Create a new console application with the name **HMACClient**, and then install the following package which will help us to issue HTTP requests to the server.

**Install-Package Microsoft.AspNet.WebApi.Client -Version 5.2.7**

You also need to include a reference to the **System.Web** dll. To do so, right-click on your project in the Solution Explorer and choose Add Reference and then search for **System.Web**and add that reference as shown in the below image.



We also need the **System.Runtime.Caching** library for storing the data in cache memory. To do so, right-click on your project in the Solution Explorer and choose Add Reference and then search for **System.Runtime.Caching** and then add that reference as shown in the below image.



**Step2: Adding the Model Classes**

In our client application, we will make use of HTTP Post request in order to show how we can include the request body within the HMAC Signature. So. add a class file with the name “**Order**” and then copy and paste the following code:

**using** *System;*

**namespace** *HMACClient*

**{**

**public** **class** Order

**{**

**public** **int** OrderID **{** **get**; **set**; **}**

**public** string CustomerName **{** **get**; **set**; **}**

**public** string CustomerAddress **{** **get**; **set**; **}**

**public** string ContactNumber **{** **get**; **set**; **}**

**public** Boolean IsShipped **{** **get**; **set**; **}**

**}**

**}**

**Step3: Call the API using HTTPClient**

Now we will see, how to use the **HTTPClient** library installed in **Step1** to issue an **HTTP Post** request to the Web API (that we are going to build in the next section i.e. in Section3) using HMAC Authentication. So open the **Program.cs** file and then copy and paste the following code:

**Note:** In the next step we will create the **HMACDelegatingHandler** which we are going to use in this step. So don’t be confused by getting the error for **HMACDelegatingHandler**.

**using** *System;*

**using** *System.Net.Http;*

**using** *System.Threading.Tasks;*

**namespace** *HMACClient*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

RunAsync**()**.Wait**()**;

Console.ReadLine**()**;

**}**

**static** **async** Task RunAsync**()**

**{**

Console.WriteLine**(**"Calling the back-end API"**)**;

//Need to change the port number

//provide the port number where your api is running

string apiBaseAddress = "http://localhost:63493/";

HMACDelegatingHandler customDelegatingHandler = new HMACDelegatingHandler**()**;

HttpClient client = HttpClientFactory.Create**(**customDelegatingHandler**)**;

var order = new Order

**{**

OrderID = 10248,

CustomerName = "Pranaya Rout",

CustomerAddress = "Mumbai|Mahatashtra|IN",

ContactNumber = "1234567890",

IsShipped = **true**

**}**;

HttpResponseMessage response = **await** client.PostAsJsonAsync**(**apiBaseAddress + "api/orders", order**)**;

**if** **(**response.IsSuccessStatusCode**)**

**{**

string responseString = **await** response.Content.ReadAsStringAsync**()**;

Console.WriteLine**(**responseString**)**;

Console.WriteLine**(**"HTTP Status: {0}, Reason {1}. Press ENTER to exit", response.StatusCode, response.ReasonPhrase**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Failed to call the API. HTTP Status: {0}, Reason {1}", response.StatusCode, response.ReasonPhrase**)**;

**}**

**}**

**}**

**}**

**Explanation of the above code:**

What we implemented in the above class is very simple. Here we are issuing an HTTP Post request to the endpoint “**/api/orders**” including the serialized order object. This endpoint (“**/api/orders**“) is protected by the Web Server using the HMAC Authentication. Finally, if the returned response status is **200 OK**, then we are printing the response returned from Web API Server.

The important thing to keep in mind is that here we are using a custom delegate handler with the name **HMACDelegatingHandler**. This is the handler which will help us to intercept the request before sending it to the Web API Server. And this is the handler where we need to write the HMAC Signature generation code and also the code that needs to attach the signature in the Authorization header.

**Step4: Implement the HTTPClient Custom Handler (HMACDelegatingHandler)**

**HTTPClient** allows us to create a custom message handler which will get created and added to the request message handlers chain. The important thing is that this handler allows us to write the logic as per our requirement. So here we need to write the logic to build the hash (**HMAC Signature**) and also the logic to set the **HMAC Signature in the Authorization header**.

To do so, create a class file with the name **HMACDelegatingHandler** and then copy and paste the following code:

**using** *System;*

**using** *System.Net.Http;*

**using** *System.Net.Http.Headers;*

**using** *System.Security.Cryptography;*

**using** *System.Text;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**using** *System.Web;*

**namespace** *HMACClient*

**{**

**public** **class** HMACDelegatingHandler : DelegatingHandler

**{**

// First obtained the APP ID and API Key from the server

// The APIKey MUST be stored securely in db or in the App.Config

**private** string APPId = "65d3a4f0-0239-404c-8394-21b94ff50604";

**private** string APIKey = "WLUEWeL3so2hdHhHM5ZYnvzsOUBzSGH4+T3EgrQ91KI=";

**protected** **async** **override** Task**<**HttpResponseMessage**>** SendAsync**(**HttpRequestMessage request, CancellationToken cancellationToken**)**

**{**

HttpResponseMessage response = **null**;

string requestContentBase64String = string.Empty;

//Get the Request URI

string requestUri = HttpUtility.UrlEncode**(**request.RequestUri.AbsoluteUri.ToLower**())**;

//Get the Request HTTP Method type

string requestHttpMethod = request.Method.Method;

//Calculate UNIX time

DateTime epochStart = new DateTime**(**1970, 01, 01, 0, 0, 0, 0, DateTimeKind.Utc**)**;

TimeSpan timeSpan = DateTime.UtcNow - epochStart;

string requestTimeStamp = Convert.ToUInt64**(**timeSpan.TotalSeconds**)**.ToString**()**;

//Create the random nonce for each request

string nonce = Guid.NewGuid**()**.ToString**(**"N"**)**;

//Checking if the request contains body, usually will be null wiht HTTP GET and DELETE

**if** **(**request.Content != **null)**

**{**

// Hashing the request body, so any change in request body will result a different hash

// we will achieve message integrity

**byte[]** content = **await** request.Content.ReadAsByteArrayAsync**()**;

MD5 md5 = MD5.Create**()**;

**byte[]** requestContentHash = md5.ComputeHash**(**content**)**;

requestContentBase64String = Convert.ToBase64String**(**requestContentHash**)**;

**}**

//Creating the raw signature string by combinging

//APPId, request Http Method, request Uri, request TimeStamp, nonce, request Content Base64 String

string signatureRawData = String.Format**(**"{0}{1}{2}{3}{4}{5}", APPId, requestHttpMethod, requestUri, requestTimeStamp, nonce, requestContentBase64String**)**;

//Converting the APIKey into byte array

var secretKeyByteArray = Convert.FromBase64String**(**APIKey**)**;

//Converting the signatureRawData into byte array

**byte[]** signature = Encoding.UTF8.GetBytes**(**signatureRawData**)**;

//Generate the hmac signature and set it in the Authorization header

**using** **(**HMACSHA256 hmac = new HMACSHA256**(**secretKeyByteArray**))**

**{**

**byte[]** signatureBytes = hmac.ComputeHash**(**signature**)**;

string requestSignatureBase64String = Convert.ToBase64String**(**signatureBytes**)**;

//Setting the values in the Authorization header using custom scheme (hmacauth)

request.Headers.Authorization = new AuthenticationHeaderValue**(**"hmacauth", string.Format**(**"{0}:{1}:{2}:{3}", APPId, requestSignatureBase64String, nonce, requestTimeStamp**))**;

**}**

response = **await** **base**.SendAsync**(**request, cancellationToken**)**;

**return** response;

**}**

**}**

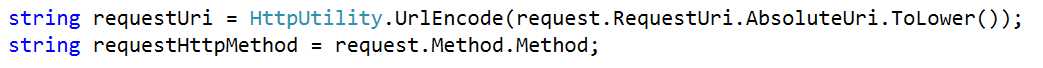
**}**

**Explanation of the above code:**

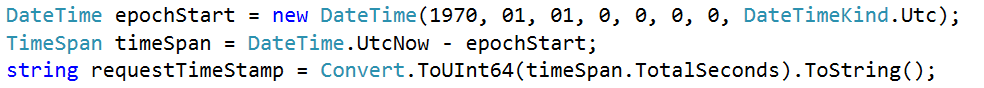
In the above example, for the simplicity of this demo and to focus of **HMAC Authentication**, we have hardcoded the **APP Id** and **API Key** values which we already obtained from the server, but in real-time, we need to store those values in some secure repository like a database or in the config file. The following code does this.

What is HMAC Authentication in WEB API?

We have got the full request URI and then we safely Encoded the URL. We need to do this because if there are any query strings sent with the request, then they will be safely encoded. Then we read the HTTP Method type from the request object, in our case, the HTTP Method type will be POST. The following code shows this



Then we have calculated the time stamp value for the request using the [UNIX timing](http://en.wikipedia.org/wiki/Unix_time" \t "_blank) (number of seconds since Jan. 1st 1970). This will help us to avoid any issues that might happen if the client and the server reside in two different time zones. The following code does this.

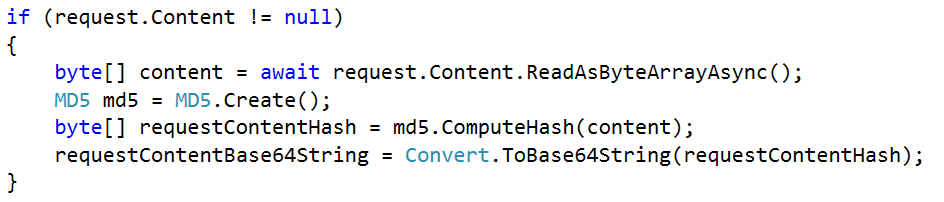


Next, we have generated a **nonce** value by using **GUID** and this value needs to be unique for each request. The following line of code does the same.

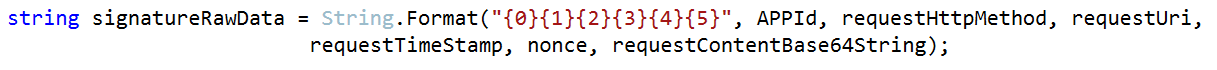
How does the HMAC Authentication work?  

Then we have checked whether the request contains a body or not. If the request type is POST or PUT, then it will contain a body. In our example the request type is POST, so it contains a body. If the request contains a body then we need to do the following things;

First, we need to use any hashing algorithm to hash the body content. Here we are using the **MD5 hashing algorithm** to hash the body content. Once the hash is generated, then we need to convert that hash into a Base64 string. The following code does the same.



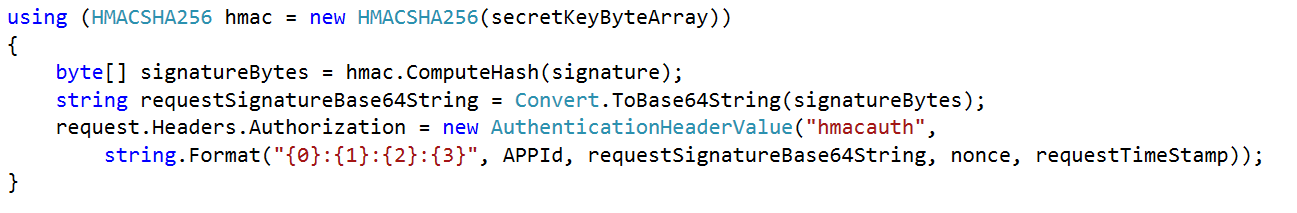
Then, we need to build the **signature raw data** by combining the parameters such as **APP Id, request HTTP Method Type, request URI, request timestamp, nonce, and requestContentBase64String** without any delimiters. The following code does this.



Then we need to convert the **signature raw data** into a **byte array.**The following code does the same.

**byte[] signature = Encoding.UTF8.GetBytes(signatureRawData);**

Finally, we have applied the hashing algorithm using the shared secret **API Key** and convert the result into the base64 format and combined the (**APPId:requestSignatureBase64String:nonce:requestTimeStamp**) using ‘:’ colon delimiter and set this combined string in the Authorization header for the request using a custom scheme named “hmacauth”. The following code does the same thing.



Notice that the nonce and the timestamp values are also included in creating the request signature as well as they are sent as plain text values so that they can be validated on the server to protect our API from replay attacks. In the latter part of this article, we will discuss what are reply attacks and how to avoid them?

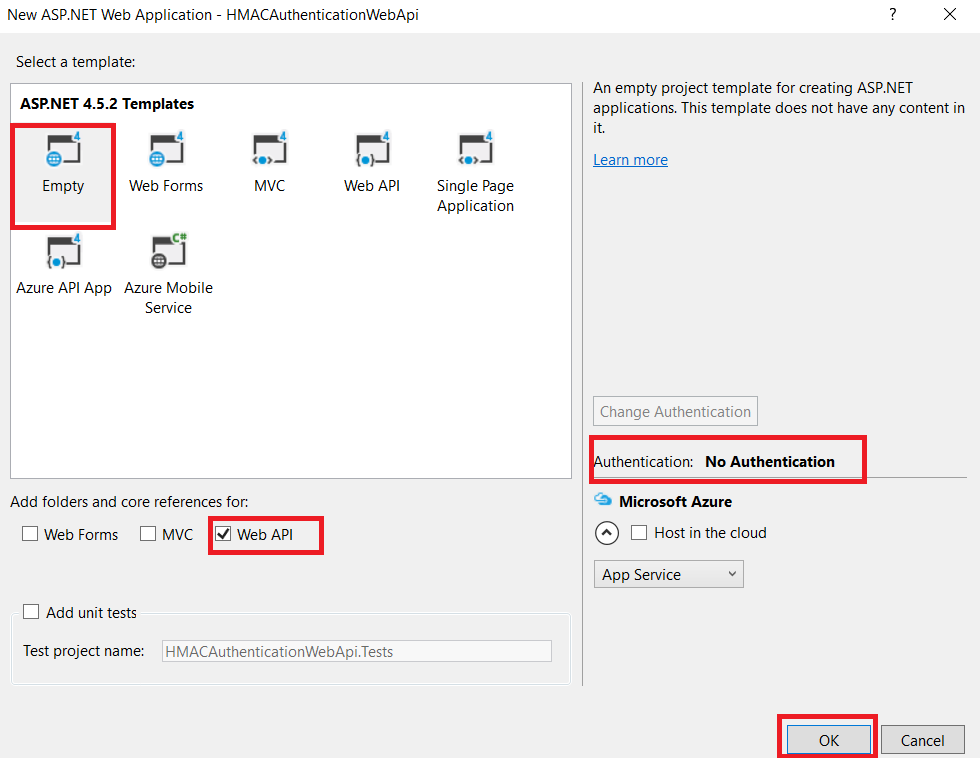
That’s it. We have done the implementation of the client application. Now let’s move to build the Web API application (server) which will be protected using the HMAC Authentication.

**Section3: Building the Server (Backend API)**

As a Web Server, here we are going to create a Web API application which will be protected using the **HMAC Authentication**. Let’s discuss the step by step procedure to implement this.

**Step1: Add an empty Web API Application**

Create an empty Web API application with the name “**HMACAuthenticationWebApi**” as shown in the image below.



**Step2: Adding Order Model**

Create a class file within the Models folder with the name **Order** which will exactly the same Orders class that we created in the Client application. So copy and paste the following code within **Order.cs** file

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *HMACAuthenticationWebApi.Models*

**{**

**public** **class** Order

**{**

**public** **int** OrderID **{** **get**; **set**; **}**

**public** string CustomerName **{** **get**; **set**; **}**

**public** string CustomerAddress **{** **get**; **set**; **}**

**public** string ContactNumber **{** **get**; **set**; **}**

**public** Boolean IsShipped **{** **get**; **set**; **}**

**public** **static** List**<**Order**>** GetOrders**()**

**{**

List**<**Order**>** OrderList = new List**<**Order**>**

**{**

new Order **{**OrderID = 101, CustomerName = "Pranaya", CustomerAddress = "Amman", ContactNumber = "9876543210", IsShipped = **true** **}**,

new Order **{**OrderID = 102, CustomerName = "Anurag", CustomerAddress = "Dubai",ContactNumber = "9876543210", IsShipped = **false}**,

new Order **{**OrderID = 103, CustomerName = "Priyanka", CustomerAddress = "Jeddah", ContactNumber = "9876543210", IsShipped = **false** **}**,

new Order **{**OrderID = 104, CustomerName = "Hina", CustomerAddress = "Abu Dhabi",ContactNumber = "9876543210", IsShipped = **false}**,

new Order **{**OrderID = 104, CustomerName = "Sambit", CustomerAddress = "Kuwait", ContactNumber = "9876543210",IsShipped = **true}**

**}**;

**return** OrderList;

**}**

**}**

**}**

The above Order class implementation is very simple and straightforward. We have one method which will return the list of orders.

**Step3: Adding Orders Web API Controller**

Here we will add an **empty Web API Controller**with the name “**Orders**” within the **Controllers** Folder and then we will create two simple HTTP methods. So, create the Orders Controller and then copy and paste the following code in it.

**using** *HMACAuthenticationWebApi.Models;*

**using** *System.Net.Http;*

**using** *System.Security.Claims;*

**using** *System.Web.Http;*

**namespace** *HMACAuthenticationWebApi.Controllers*

**{**

**[**RoutePrefix**(**"api/Orders"**)]**

**public** **class** OrdersController : ApiController

**{**

**[**Route**(**""**)]**

**public** IHttpActionResult Get**()**

**{**

**return** Ok**(**Order.GetOrders**())**;

**}**

**[**Route**(**""**)]**

**public** IHttpActionResult Post**(**Order order**)**

**{**

**return** Ok**(**order**)**;

**}**

**}**

**}**

In the above Web API Controller, we are not doing anything special, just a basic Web API controller which is not protected and allows anonymous calls. Later we will discuss how to protect this.

**Step4: Build the HMAC Authentication Filter**

Here we will add a **Custom Authentication Filter** (a class which is inherited from the **IAuthenticationFilter**) and within that Custom Filter, we need to write the logic to re-generating the HMAC signature from the request data and Authorization header and then  we also need to write the logic compare the generated HAMC Signature with the signature received from the client**.**

The **Authentication Filter** is available from **Web API 2** and we need to use this filter only for the authentication purposes. In our case, we will use this custom Authentication Filter to write the validation logic which will validate the authenticity of the signature received from the client.

The important thing about this filter is that it runs before any other filters especially the authorization filter runs. Create a class file with the name **HMACAuthenticationAttribute** within the **Models** folder and then copy and paste the following code.

**Note:** You will get a compiler error for **IsValidRequest** and **ResultWithChallenge** method that we will implement in our next step.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Net.Http;*

**using** *System.Net.Http.Headers;*

**using** *System.Runtime.Caching;*

**using** *System.Security.Cryptography;*

**using** *System.Security.Principal;*

**using** *System.Text;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**using** *System.Web;*

**using** *System.Web.Http.Filters;*

**using** *System.Web.Http.Results;*

**namespace** *HMACAuthenticationWebApi.Models*

**{**

**public** **class** HMACAuthenticationAttribute : Attribute, IAuthenticationFilter

**{**

**private** **static** Dictionary**<**string, string**>** allowedApps = new Dictionary**<**string, string**>()**;

**private** **readonly** UInt64 requestMaxAgeInSeconds = 300; //Means 5 min

**private** **readonly** string authenticationScheme = "hmacauth";

**public** HMACAuthenticationAttribute**()**

**{**

**if** **(**allowedApps.Count == 0**)**

**{**

allowedApps.Add**(**"65d3a4f0-0239-404c-8394-21b94ff50604", "WLUEWeL3so2hdHhHM5ZYnvzsOUBzSGH4+T3EgrQ91KI="**)**;

**}**

**}**

**public** Task AuthenticateAsync**(**HttpAuthenticationContext context, CancellationToken cancellationToken**)**

**{**

var req = context.Request;

**if** **(**req.Headers.Authorization != **null** && authenticationScheme.Equals**(**req.Headers.Authorization.Scheme, StringComparison.OrdinalIgnoreCase**))**

**{**

var rawAuthzHeader = req.Headers.Authorization.Parameter;

var autherizationHeaderArray = GetAutherizationHeaderValues**(**rawAuthzHeader**)**;

**if** **(**autherizationHeaderArray != **null)**

**{**

var APPId = autherizationHeaderArray**[**0**]**;

var incomingBase64Signature = autherizationHeaderArray**[**1**]**;

var nonce = autherizationHeaderArray**[**2**]**;

var requestTimeStamp = autherizationHeaderArray**[**3**]**;

var isValid = IsValidRequest**(**req, APPId, incomingBase64Signature, nonce, requestTimeStamp**)**;

**if** **(**isValid.Result**)**

**{**

var currentPrincipal = new GenericPrincipal**(**new GenericIdentity**(**APPId**)**, **null)**;

context.Principal = currentPrincipal;

**}**

**else**

**{**

context.ErrorResult = new UnauthorizedResult**(**new AuthenticationHeaderValue**[**0**]**, context.Request**)**;

**}**

**}**

**else**

**{**

context.ErrorResult = new UnauthorizedResult**(**new AuthenticationHeaderValue**[**0**]**, context.Request**)**;

**}**

**}**

**else**

**{**

context.ErrorResult = new UnauthorizedResult**(**new AuthenticationHeaderValue**[**0**]**, context.Request**)**;

**}**

**return** Task.FromResult**(**0**)**;

**}**

**public** Task ChallengeAsync**(**HttpAuthenticationChallengeContext context, CancellationToken cancellationToken**)**

**{**

context.Result = new ResultWithChallenge**(**context.Result**)**;

**return** Task.FromResult**(**0**)**;

**}**

**public** **bool** AllowMultiple

**{**

**get** **{** **return** **false**; **}**

**}**

**private** string**[]** GetAutherizationHeaderValues**(**string rawAuthzHeader**)**

**{**

var credArray = rawAuthzHeader.Split**(**':'**)**;

**if** **(**credArray.Length == 4**)**

**{**

**return** credArray;

**}**

**else**

**{**

**return** **null**;

**}**

**}**

**}**

**}**

**Explanation of the Above Code:**

The above **HMACAuthenticationAttribute** class is derived from the **Attribute** class. So we can use this **HMACAuthenticationAttribute** class as a Filter Attribute over the controllers or HTTP action methods.

The constructor of the **HMACAuthenticationAttribute** class currently filled a dictionary named “**allowedApps**”. We did this only for the demo purpose. In real-time, we need to store the **Public Shared APP Id** and **Private Secret API Key** in some secure repository like a database or config file.

In the “**AuthenticateAsync**” method, we implement the logic for validating the signature of the incoming request.

Next, we need to make sure that the **Authorization Header** is present in the request and it should not be empty. We also need to make sure that it contains the “**hmacauth**” scheme. If everything is fine, then we need to read the **Authorization Header** value from the request and then split its content based on the delimiter we have specified earlier in client i.e. using a colon “:”.

Finally, we are calling the “**IsValidRequest**” method where we implement all the logic of reconstructing the HMAC Signature from the request data and then comparing this signature with the incoming signature. We will implement this method in **Step6** of this section.

In case the **Authorization Header** is not present or if the Authorization Header does not the “**hmacauth**” scheme or if the “**IsValidRequest**” method returns false, then we will consider this request as an unauthorized request and returns **401 unauthorized.** We should return an authentication challenge to the response, and this should be implemented within the method “**ChallengeAsync**” which we will implement in the next step.

**Step5: Adding the authentication challenge to the response**

To add the authentication challenge to the unauthorized response, create a class file with the name ResultWithChallenge within the Models folder and then copy and paste the following code.

**using** *System.Net;*

**using** *System.Net.Http;*

**using** *System.Net.Http.Headers;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**using** *System.Web.Http;*

**namespace** *HMACAuthenticationWebApi.Models*

**{**

**public** **class** ResultWithChallenge : IHttpActionResult

**{**

**private** **readonly** string authenticationScheme = "hmacauth";

**private** **readonly** IHttpActionResult next;

**public** ResultWithChallenge**(**IHttpActionResult next**)**

**{**

this.next = next;

**}**

**public** **async** Task**<**HttpResponseMessage**>** ExecuteAsync**(**CancellationToken cancellationToken**)**

**{**

var response = **await** next.ExecuteAsync**(**cancellationToken**)**;

**if** **(**response.StatusCode == HttpStatusCode.Unauthorized**)**

**{**

response.Headers.WwwAuthenticate.Add**(**new AuthenticationHeaderValue**(**authenticationScheme**))**;

**}**

**return** response;

**}**

**}**

**}**

The above code is very simple. Basically here we add the “**WWW-Authenticate**” header to the response using our “**hmacauth**” custom scheme.

**Step6: Implementing the IsValidRequest Method**

The custom implementation logic of reconstructing the signature and comparing it with the signature received from the client is done here. So let’s add the code first and then we will discuss what this method is responsible for. Open the file “**HMACAuthenticationAttribute.cs”** which is present inside the **Models** Folder and then paste the following code below the **GetAutherizationHeaderValues** method.

**private** **async** Task**<bool>** IsValidRequest**(**HttpRequestMessage req, string APPId, string incomingBase64Signature, string nonce, string requestTimeStamp**)**

**{**

string requestContentBase64String = "";

string requestUri = HttpUtility.UrlEncode**(**req.RequestUri.AbsoluteUri.ToLower**())**;

string requestHttpMethod = req.Method.Method;

**if** **(**!allowedApps.ContainsKey**(**APPId**))**

**{**

**return** **false**;

**}**

var sharedKey = allowedApps**[**APPId**]**;

**if** **(**isReplayRequest**(**nonce, requestTimeStamp**))**

**{**

**return** **false**;

**}**

**byte[]** hash = **await** ComputeHash**(**req.Content**)**;

**if** **(**hash != **null)**

**{**

requestContentBase64String = Convert.ToBase64String**(**hash**)**;

**}**

string data = String.Format**(**"{0}{1}{2}{3}{4}{5}", APPId, requestHttpMethod, requestUri, requestTimeStamp, nonce, requestContentBase64String**)**;

var secretKeyBytes = Convert.FromBase64String**(**sharedKey**)**;

**byte[]** signature = Encoding.UTF8.GetBytes**(**data**)**;

**using** **(**HMACSHA256 hmac = new HMACSHA256**(**secretKeyBytes**))**

**{**

**byte[]** signatureBytes = hmac.ComputeHash**(**signature**)**;

**return** **(**incomingBase64Signature.Equals**(**Convert.ToBase64String**(**signatureBytes**)**, StringComparison.Ordinal**))**;

**}**

**}**

**private** **bool** isReplayRequest**(**string nonce, string requestTimeStamp**)**

**{**

**if** **(**System.Runtime.Caching.MemoryCache.Default.Contains**(**nonce**))**

**{**

**return** **true**;

**}**

DateTime epochStart = new DateTime**(**1970, 01, 01, 0, 0, 0, 0, DateTimeKind.Utc**)**;

TimeSpan currentTs = DateTime.UtcNow - epochStart;

var serverTotalSeconds = Convert.ToUInt64**(**currentTs.TotalSeconds**)**;

var requestTotalSeconds = Convert.ToUInt64**(**requestTimeStamp**)**;

**if** **((**serverTotalSeconds - requestTotalSeconds**)** **>** requestMaxAgeInSeconds**)**

**{**

**return** **true**;

**}**

System.Runtime.Caching.MemoryCache.Default.Add**(**nonce, requestTimeStamp, DateTimeOffset.UtcNow.AddSeconds**(**requestMaxAgeInSeconds**))**;

**return** **false**;

**}**

**private** **static** **async** Task**<byte[]>** ComputeHash**(**HttpContent httpContent**)**

**{**

**using** **(**MD5 md5 = MD5.Create**())**

**{**

**byte[]** hash = **null**;

var content = **await** httpContent.ReadAsByteArrayAsync**()**;

**if** **(**content.Length != 0**)**

**{**

hash = md5.ComputeHash**(**content**)**;

**}**

**return** hash;

**}**

**}**

**Explanation of the Above Code:**

First, we check whether the received **Public Shared APP ID** is registered in our system or not, if it is not registered in our system, then we simply return **false** from the isValidRequest method. But if we found the **Public Shared APP ID** in our system, then we need to check whether the request received is a **replay request.**

**What is** R**eplay Request?**

The **replay request** means we need to check if the **nonce** received from the client is used before. Currently, we are storing all the **nonce** received from the client in **Cache Memory** for 5 minutes only. Here we are using the Runtime Caching.

Let us understand the **replay request** with an example. For example, if the client generates a **nonce** lets say “**abcd1234**” and send it with the request to the server. Then the server will check whether this nonce “**abcd1234**” is used before. If not then the server will store the **nonce** value in **Cache Memory** for the next 5 minutes. So any request coming from the client with the same **nonce** value i.e. “**abcd1234**” during the 5 minutes time interval will be considered as a **replay attack** or **replay request**. if the same **nonce** “**abcd1234**” is used after 5 minutes time interval then this is fine and the request is not considered as a **replay request**.

But there might a situation where let’s say an evil person might try to re-post the same request using the same nonce after the 5 minutes window, so in situation like this the **request timestamp**becomes handy, the implementation here is comparing the current server UNIX time with the request UNIX time received from the client, if the request timestamp is older than 5 minutes then it is rejected by the server and the evil person has no possibility to fake the request timestamp and send fresher one because we have already included the request timestamp in the signature raw data, so any changes on it will result in a new signature and it will not match the client incoming signature.

In the last step, we need to compute the MD5 hash of the body content if it is available (POST or PUT methods), then we built the signature raw data by concatenating the parameters (APP ID, request HTTP method, request URI, request timestamp, nonce, requestContentBase64String) without any delimiters. It is mandatory that both the parties (Client and Server) need to use the same data format to produce the same signature; the data eventually will get hashed using the same hashing algorithm and API Key used by the client. If the incoming client signature equals the signature generated on the server then we will consider this request as authentic and will process it.

**Step7: Securing the API End Points:**

The final thing here we need to do is to add the HMACAuthentication attribute to the controller actions so that the action is protected from the anonymous access. So open the Orders controller and add the attribute “**HMACAuthentication**” as shown below:

**using** *HMACAuthenticationWebApi.Models;*

**using** *System.Web.Http;*

**namespace** *HMACAuthenticationWebApi.Controllers*

**{**

**[**RoutePrefix**(**"api/Orders"**)]**

**public** **class** OrdersController : ApiController

**{**

**[**Route**(**""**)]**

**[**HMACAuthentication**]**

**public** IHttpActionResult Get**()**

**{**

**return** Ok**(**Order.GetOrders**())**;

**}**

**[**Route**(**""**)]**

**[**HMACAuthentication**]**

**public** IHttpActionResult Post**(**Order order**)**

**{**

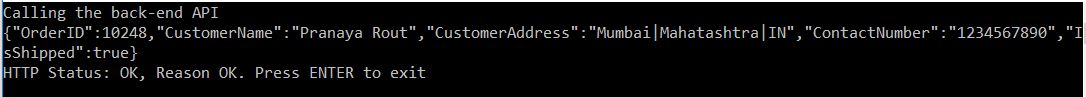
**return** Ok**(**order**)**;

**}**

**}**

**}**

That’s it. We have done with our implementation. So first the run the Web API and application and figure out the port number on which the application is running. Once you figure out the Port number on which your API is running, you need to update the above port number with API base address port number in the Client Application. Once you update the Port number, then run the client application and see everything is working as expected as shown in the below image.



In this article, we discussed the HMAC authentication with the HTTP Request and in the next article, we will discuss how to use the HMAC authentication with the HTTP Response.